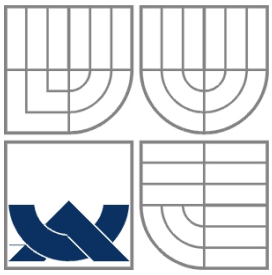
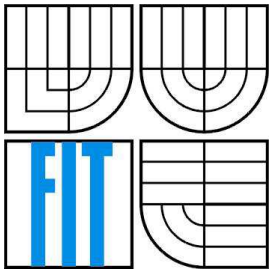


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DATABÁZOVÁ NEZÁVISLOST JÁDRA SYSTÉMU PRO DOLOVÁNÍ Z DAT FIT-MINER

DATA INDEPENDENCY OF THE FIT-MINER DATA MINING SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Ondřej Novák

VEDOUCÍ PRÁCE
SUPERVISOR
Doc. Ing. Jaroslav Zendulka, CSc.

BRNO 2013

Abstrakt

Systém pro dolování z dat FIT-Miner je nyní závislý pouze na jednom specifickém SŘBD. Tato diplomová práce se zabývá analýzou implementace pracující s databází, jednotlivých modulů a operacemi pro dolování z dat. Poté návrhem změn systému FIT-Miner tak, aby byl schopen pracovat s více SŘBD. A nakonec popisem implementace těchto změn.

Abstract

System for data mining Fit-Miner is now dependant on only one specific DBMS. This master's thesis deals with analysis of implementation that works with database, modules and functions for data mining. Next it shows the set of changes which will allow FIT-Miner to work with another DBMS. And finally, a description of the implementation of these changes.

Klíčová slova

dolování z dat, DMSL, NetBeans, Oracle Data Mining, FIT-Miner, systém pro dolování z dat, databáze, MySQL, PosgreSQL, Oracle,

Keywords

Data mining, DMSL, NetBeans, Oracle Data Mining, FIT-Miner, datamining systém, database, MySQL, PosgreSQL, Oracle

Citace

Novák Ondřej: Databázová nezávislost jádra systému pro dolování z dat FIT-Miner, diplomová práce, Brno, FIT VUT v Brně, 2013

Databázová nezávislost jádra systému pro dolování z dat FIT-Miner

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Jaroslava Zendulky, CSc.
Další informace mi poskytli Ing. Martin Hlosta a Ing. Michal Šebek.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ondřej Novák
22. 5. 2013

Poděkování

Chtěl bych poděkovat panu Doc. Ing. Jaroslavu Zendulkovi, CSc. za jeho pomoc, konzultace a podporu při řešení této diplomové práce. Také Ing. Marinu Hlostovi za poskytnutí informací o Fit-Miner systému.

Obsah

1 Úvod

Systém FIT Miner, vyvíjený na naší fakultě, slouží k dolováním z dat z databázového systému.

Dolování z dat je v dnešní době stále se rozšiřující oblastí, která s rostoucím množstvím ukládaných informací a nároky na jejich masové zpracování získává vekou pozornost. Náplní dolování z dat je extrakce zajímavých modelů dat a vzorů, které jsou netriviální, skryté, dříve neznáme a potencionálně užitečné. Nemělo by jít o informace získatelné jednoduchými postupy přes dotazy do databáze, ale o data získané sofistikovaným netriviálním postupem.

Za dobu vývoje prošel systém FIT-Miner spoustou změn, při kterých byla snaha vytvořit systém více robustní a přehledný. FIT-Miner se skládá z jádra systému a jednotlivých dolovacích modulů. Nicméně nynější implementace nedovoluje použít jiný systém řízení báze dat než Oracle.

Cílem této diplomové práce je tedy zavést možnost použití jiného systému řízení báze dat. Aby to bylo možné, je prvně potřeba analyzovat veškerý kód jádra, který se nějakým způsobem podílí na komunikací s SŘBD a navrhnout implementovatelné změny tak, aby bylo možné jej využívat nezávisle na použitém SŘBD.

Některé funkčnosti, které systém Oracle zpřístupňuje v rámci své Oracle Data Mining (ODM) implementace nemusí a zřejmě ani nebudou implementovány v jiných SŘBD a je tedy nutné všechny takovéto případy v rámci hlavní části systému změnit a vytvořit vlastní implementaci, která by byla nezávislá.

Aby mohl být FIT-Miner reálně použit s jinými SŘBD je také potřeba zjistit, zda a jak komunikují jednotlivé moduly s ODM. Ty moduly, které komunikují přímo, bude vhodné také analyzovat, navrhnout úpravy a nejlépe sjednotit volání do databáze přes jednotné rozhraní hlavní části systému.

Spolu s analýzou, návrhem a implementací změn v rámci již existují implementace v systému FIT-Miner se budu zabývat i zprovozněním pro dva nové SŘBD. Půjde o MySQL a PostgreSQL.

Práce je rozdělena na několik části. Kapitola 2 obsahuje obecné informace o dolování z dat a částí tohoto procesu. Kapitola 3 poté zahrnuje informace o koncepci FIT-Miner systému, popis jazyka DMSL jakožto hlavní součástí pro popis dolovacích úloh a procesů. Taktéž je zde popsána struktura jádra FIT-Miner systému a napojení modulů. Kapitola 4 je věnována knihovně Weka, která se bude používat na nahrazení částí Oracle implementace. Kapitola 5 se věnuje analýze současného řešení a to jak v rámci hlavní části systému FIT-Miner, tak i v rámci jednotlivých modulů a zvažuje možnosti změn.

Kapitola 6 je částí popisující stručně koncepční změny v systému, které bylo potřeba provést, aby byla možná databázová nezávislost a bylo možné používat dvě nové navrhované SŘBD.

Kapitola 7 se zabývá vlastním návrhem a implementací systému, popisuje změny v hlavní části systému i v modulech, tak jak byly provedeny v rámci této diplomové práce. Taktéž popisuje vznik nových částí pro implementaci dvou nových ŠRBD. Kapitola 8 popisuje testování nových změn v systému. Kapitola 9 obsahuje návod pro přidání dalších SŘBD. Kapitola 10 zhodnocuje přínos této práce, výsledné řešení a možný další vývoj aplikace FIT-Miner.

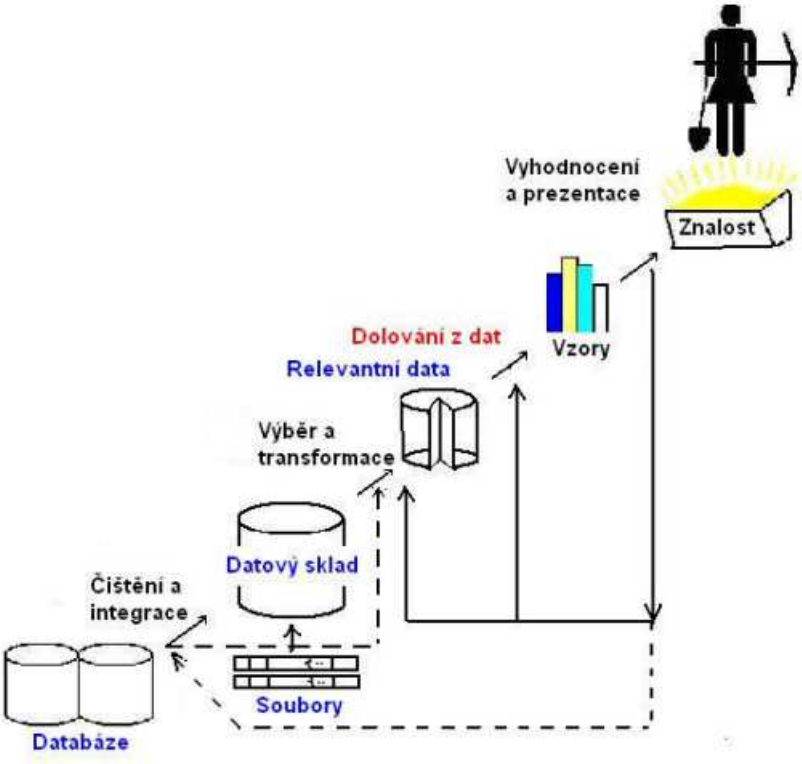
2 Dolování z dat

V této kapitole jsem čerpal především z [13][12].

Jelikož se tato diplomová práce nezabývá primárně dolováním z dat, ale implementačním detailům systému FIT-Miner, nebudu se zde o dolovacím procesu rozepisovat do detailů. Nicméně i tak je důležité pochopit strukturu a problematiku dolování z dat při analýze FIT-Miner systému. Pro dolování z dat existuje ještě jeden výraz, kterým je „získávání znalostí z databází“. Prakticky jde o synonymum, ač je to možná spíše trochu obecnější pojem a dolování z dat občas označuje až samotnou část procesu získávání znalostí. V této publikaci je ovšem pro jednoduchost budeme chápat jako synonyma.

Můžeme říci, že získávání znalostí z databází je extrakce (neboli „dolování“) zajímavých (netriviálních, skrytých, dříve neznámých a potenciálně užitečných) modelů dat a vzorů velkých objemů dat. Tyto modely a vzory reprezentují znalosti získané z dat. [12]. Jak je z této citace zřejmé, jednoduché dotazy do databáze nejsou dolováním z dat. Pro dolování z dat je nutné, aby přístup k datům a výsledkům z nich byl sofistikovaný.

Samotný proces dolování z dat, zobrazený na obrázku Obrázek 2-1 je rozdělen na několik dílčích kroků, které lze seskupit do dvou celků. Prvním je předzpracování dat a druhým již vlastní „dolování z dat“.



Obrázek 2-1Proces získávání znalostí, převzato z [12]

2.1 Předzpracování dat

Předzpracování dat je první část celého procesoru. Předzpracování dat se taktéž dělí na několik kroků.

- **Čištění dat**. Data mohou obsahovat různé nekonzistence, nebo mohou být i chybná. Cílem čištění dat je tedy odstranit chybějící nebo odlehlé hodnoty a řešit nekonzistenci dat. Tímto procesem získáváme kvalitnější, přesnější data, která poté dávají lepší výsledky v rámci vlastního dolování z dat.

- **Integrace dat** z několika různých zdrojů. Může se jednat jak o různá uložiště, tak o jednotlivé tabulky v rámci jedné databáze. Souvisí s tímto řada problému, například konflikty identifikace, nebo duplicitní údaje.

- **Transformace a výběr relevantních dat.** Data se do požadovaného tvaru pro specifickou dolovací úlohu transformují. Může se jednak například o nějaký druh normalizace hodnot, generalizaci, nebo také o vyhlazení dat. Z transformačních operací je nejzajímavější normalizace, která umí převádět hodnoty na omezené intervaly, například <0.0,1.0>.

- **Redukce dat.** Podstatou redukce dat je zmenšení objemu dat tak, aby dolovací proces nemusel pracovat s příliš velkými objemy dat. Při redukci je nutné zachovat integritu dat a jejich původní charakter, tak aby výsledky dolování nad velkým i menším objemem dat byly přibližně stejné. Pro redukci dat lze použít diskretizaci, redukci počtu hodnot nebo dimenze a spoustu dalších technik.

2.2 Dolování z dat

Po fázi předzpracování následuje vlastní část dolování z dat. Tento proces je samostatně definovaný dolovací úlohou a jejím typem a je tedy vždy velmi specifický.

V rámci dolování je třeba získání vzorů dat nebo pravidel. Může jich být velké množství, takže je potřeba tyto výsledky vyhodnotit. Zajímavé a užitečné vzory pak můžeme nazvat znalostí, kterou jsme získali.

Existují čtyři posuzující kritéria, které rozhodují, zda je vzor zajímavý či nikoliv.

Srozumitelnost, to znamená, že z něj získáváme nějakou informaci, kterou umíme interpretovat.

Platnost. Je tedy validní pro nová data.

Užitečnost takovéhoto vzoru. Pokud získáme zajímavou informaci, ale k ničemu nám nebude, nebude se jednat o znalost.

Novost. Znalost musí vždy přinášet nějaké nové poznatky a ne pouze to, co již víme.

Nakonec se **prezentují znalosti**, získané pomocí dolování z dat. Tyto znalosti je nutné pochopit a vhodně interpretovat

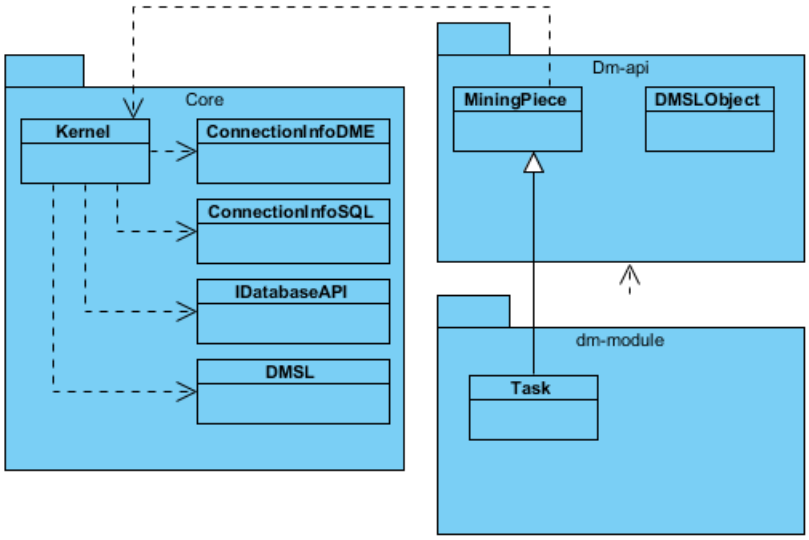
3 FIT-Miner

FIT-Miner je systém, vyvíjený především v rámci diplomových prací na FIT VUT v Brně. Jedná se o Java aplikaci, která se zaobírá dolováním z dat. V rámci aplikace se řeší příprava a definice dat, jejich vizualizace a nastavení dolovacího procesu. Vlastní dolování poté probíhá částečně na databázovém serveru a částečně v rámci implementovaných algoritmů v aplikaci a přidružených knihovnách.

3.1 Koncepce systému

Systém FIT-Miner je vystavěn nad platformou NetBeans, což je komplexní Open Source projekt od firmy Sun Microsystems, který obsahuje jak vývojové prostředí NetBeans, tak vývojovou platformu NetBeans. Z platformy se využívá v rámci projektu především grafická nadstavba.

Systém FIT-Miner je navržen jako modulární. Existuje jádro systému a několik pomocných modulů – například modul *Editor*, *File Support* nebo *API* modul. Tyto části se starají o vlastní fungování aplikace a zajišťují funkcionalitu pro ostatní funkční moduly. Druhou důležitou částí jsou dolovací moduly, implementující jednotlivé druhy dolovacích úloh a algoritmů.

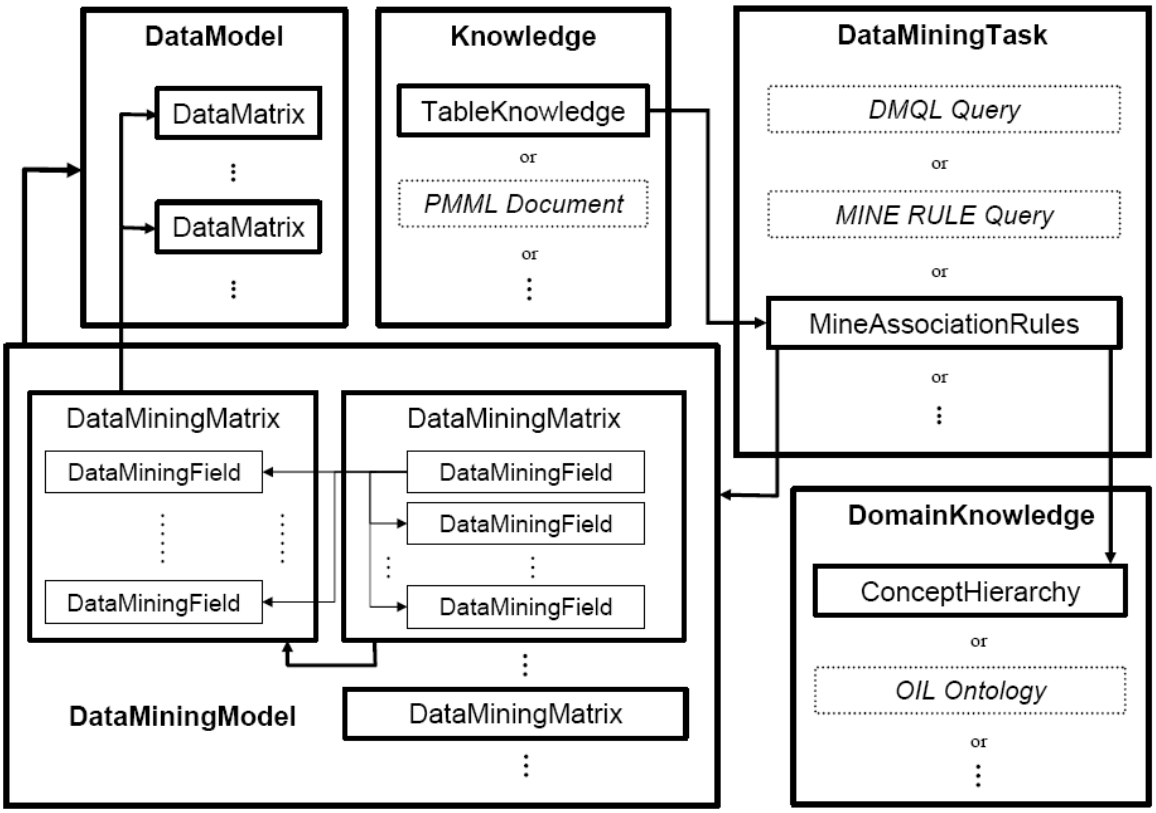


Obrázek 3-1 Závislosti mezi jádrem a moduly

Jak je z diagramu na obrázku Obrázek 3-1 patrné, jednotlivé moduly mají vždy minimálně jednu třídu odvozenou od třídy *MiningPiece*y, která má naopak zase vazbu na *Kernel* objekt. Tímto způsobem se každý modul dokáže přes *Kernel* objekt dostat k připojení k databázi – třída *ConnectionInfoSQL*, případně *ConnectionInfoDME*, k databázovému rozhraní *IDatabaseAPI* a také k vlastní definici DMSL.

3.2 Jazyk DMSL

V této části jsem vycházel z informací z [13] a [1].
DMSL (Data Mining Specification Language) je jazyk specifikující dolovací proces. Zapisuje se ve formě XML dokumentu, kde je rozdělen do několika základních sekcí popisujících postupně celý dolovací proces od vstupních dat, přes nastavení dolovací úlohy až po znalosti, respektive výstupy z dolování. Celkový přehled závislostí DMSL struktury je na schématu Obrázek 3-2.



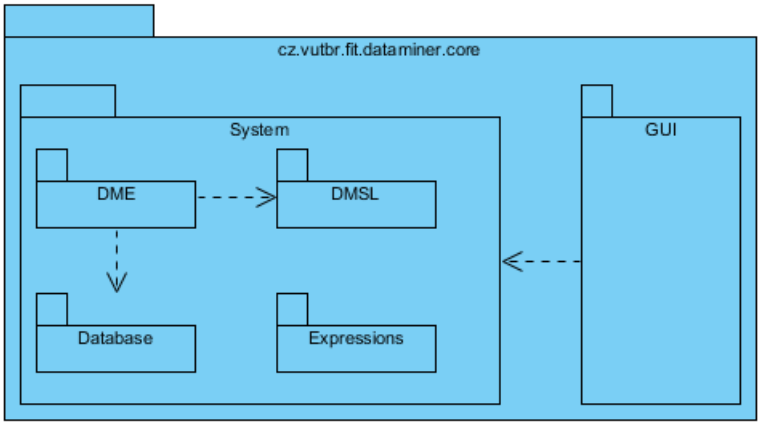
Obrázek 3-2 Závislosti DMSL elementů, převzato z [13]

DMSL lze rozdělit na pět hlavních částí, které jsou dobře viditelné na přecházejícím schématu.

1. **Data Model** - Reprezentuje vstupní data pro dolovací úlohu.
2. **Data Mining Model** – Je rozšířením data modelu. Obsahuje transformace, kterými byla upravena původní data.
3. **Domain Knowledge –** Doménové znalost, mohou být použity dolovacími úlohami při provádění dolování.
4. **Data Mining Task** – Specifikuje, jaká data mají být dolována pro jednotlivé typy znalostí.
5. **Knowledge** – Výsledek dolování z dat.

3.3 Jádro

Jádro systému existuje jako samostatný NetBeans projekt, nemá tedy vazbu na žádné jiné moduly v rámci FIT-Miner sytému. Jádro samotné je rozděleno na několik balíčků, které se starají o základní funkčnost celého systému, jak je patrné z následujícího diagramu.



Obrázek 3-3 Diagram balíčků jádra

System
Hlavní část, obsahuje třídy pro uchovávání připojení k databázi a informace o těchto připojení. Také obsahuje třídu obsahující definici vnitřních typů, používaných v rámci FIT-Miner systému. Nakonec je zde třída *Kernel*, která je velice důležitá, jelikož pro jednotlivé moduly zprostředkovává připojení do databáze, vlastní DMSL definici projektu a další užitečné věci. Každý projekt v rámci FIT-Miner systému má svůj *Kernel* objekt.

DME
Tato část jádra se soustřeďuje na transformaci dat. Obsahuje třídy na vytvoření základních konstrukcí pro diskretizaci, normalizaci, vyhlazování a ořezávání dat. Jejich účelem je předzpracování dat, například jaké sloupce a tabulky se zpracují a jaké ne, pro vlastní provedení těchto operací v rámci databázového rozhraní.

Database
Tato část jádra nás bude v rámci této diplomové práce zajímat jako jedna z nejvíce. Obsahuje rozhraní *IDatabaseAPI*, které musí být implementované pro všechny databázové systémy schopné pracovat s FIT-Miner systémem. Zatím jediná implementace je pro databázi Oracle ve zvláštním modulu. Dále se tu nachází i třídy pro uchovávání metadat a dat načtených z tabulek a sloupců databáze.

DMSL
DMSL balíček, jak již název napovídá, obsahuje třídy vztahující se k definování dolovacích úloh. Existuje zde velký počet tříd, z nich velká většina jsou obálky pro korespondující elementy v rámci DMSL. Tyto třídy kromě uchovávání hodnot a nastavení umějí i vytvářet DMSL kód pro úpravu DMSL definici. Důležitou třídou je zde přímo třída *DMSL* která definuje hlavní DMSL dokument a umí nad ním pracovat.

Expressions
Poslední částí jsou výrazy. Třídy obsažené v tomto balíčku umí definovat výrazy a měnit jejich různé notace. Například z prefixového na infixový. Objevuje se zde i třída pro zpracování výrazů funkcí s parametry, která se používá pro zpracování uživatelských funkcí z DMSL definice. Tyto třídy taktéž umí generovat DMSL kód pro jejich zápis.

3.4 Moduly

Tato část obsahuje popis jednotlivých modulů systému FIT-Miner tak, jak existují v rámci aplikace. Moduly můžeme rozdělit na dva typy. Prvním jsou pomocné moduly, které se starají například o grafy, práci s DMSL, poskytují API pro ostatní moduly a podobně. Druhá skupin jsou moduly, které implementují vlastní dolovací úlohy.

Dm-actions
Modul který odchytává akce pro spuštění celého dolování z GUI aplikace a požadavky na obnovení připojení do databáze. Nicméně to vypadá, že se nejspíše již příliš nepoužívá a dolování se spouští vlastním vybráním komponenty.
Dm-api
Důležitá část, poskytující rozhraní mezi jádrem a dalšími moduly. Obsahuje rozhraní pro práci s DMSL objekty, *DMSLObject* a několik dalších rozhraní pro editor DMSL struktury. Pak především abstraktní třídu *MiningPiece*, která se používá jako základ pro vytvoření nového dolovacího modulu, aby měl přístup do jádra systému. Taktéž obsahuje třídu pro načítání informací o databázi z properties souborů.
Dm-editor
Hlavní část GUI aplikace, kde existuje editor, palety s komponenty, nadefinované akce a další částí vztahující se ke grafickému rozhraní. Také je zde třída pro import CSV souborů.
Dm-file-support
Tento modul pracuje s DMSL soubory. Čte, zpracovává a zapisuje je na disk. Kromě toho je zde důležitá třída *DMSLDataObject*, která vytváří ke každému objektu *Kernel* z jádra aplikace a také připojení do databáze.
Dm-projects
Modul se stará o celé projekty. Obsahuje formuláře pro vytvoření nového prázdného dolovacího projektu. Také se stará o ukládání a načítání jednotlivých projektů.
Dm-ora-dbconnector
DbConnector. Obsahuje jedinou třídu implementující *IDatabaseAPI* rozhraní pro Oracle na komunikaci s databází.

Následující moduly jsou dolovací, tedy již všechny implementují nějakou dolovací úlohu.

Dm-module-ad
Modul implementuje analýzu odlehlých objektů (detekce anomálií). Vyhledává tedy hodnoty, které se výrazně odlišují od ostatních a mohou představovat zajímavé vzory.

Dm-module-ar
Implementace dolovací úlohy založené na dolování asociačních pravidel. Hledá zajímavé vzory asociací nebo korelací nad velkými množinami dat.

Dm-module-ca
Zde se řeší shluková analýza, což je proces, v kterém se rozdělí objekty do tříd na základě své podobnosti.

Dm-module-cc
V rámci tohoto modulu existuje algoritmus porovnávání výsledku shlukování. Navazuje tedy na předchozí modul.

Dm-module-dt
Klasifikační modul, který používá metodu rozhodovacího stromu.

Dm-module-es
Obsahuje implementaci dolovací úlohy pro predikci v časových řadách.

Dm-module-ga
Modul se zabývá dolovací úlohou genetického algoritmu.

Dm-module-multi
Obsahuje dolovací úlohu založenou na víceúrovňových asociačních pravidlech.

Dm-module-nb
Tento modul umí vytvořit dolovací úlohu založenou na Bayesovské klasifikaci. Jde o metodu, která má základ ve statistice.

Dm-module-NN
Další modu implementující klasifikaci. Tentokrát jde o klasifikaci pomocí neuronové sítě Backpropagation. Jde o síť, která se skládá z několika vrstev neuronů.

Dm-module-svm
Modul implementuje dva typy dolovací úlohy pro regresi, a to přesněji SVM a GLM.

Dm-module-test
Zde není implementován žádný speciální algoritmus, jde jen o jakýsi zkušební modul.

Dm-*-wrapper
Několik modulů je nazvaných jako „wrapper“, jsou to moduly, které pouze zaobalují knihovny, aby se daly použít v rámci FIT-Miner systému a nemají žádnou další funkčnost, kromě zprostředkování těchto knihoven zbytku aplikace.

3.5 Databázové závislosti

Systém FIT-Miner je nyní navrhnut tak, že veškerá komunikace s databází jde přes připojení, které je po vytvoření uchováno v třídě *Kernel*. Taktéž všechny části jádra by měly volat pouze operace přes rozhraní *IDatabaseAPI*, které je implementováno v modulu *dm-ora-dbconnector*. Toto rozhraní, respektive třída, která ho implementuje, je také po vytvoření přístupna z třídy *Kernel*.

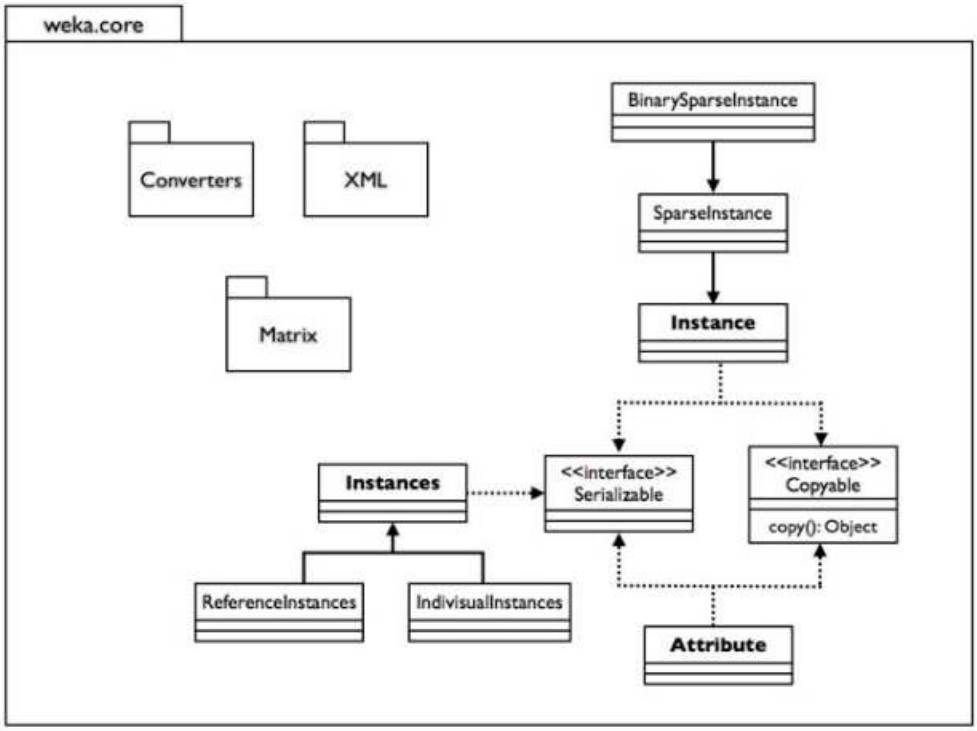
Moduly již nejsou tolik odstíněné od databáze rozhraním tak, jako jádro, nicméně pořád používají pouze připojení, které jim předá jádro systému. Mohou ovšem používat vlastní dotazy do databáze, což je trochu nevhodné při použití různých SŘBD. Navíc používají z velké části připojení na DME, místo klasického SQL, které v případě potřeby dostanou právě ze zmíněného DME připojení.

4 Knihovna Weka

Weka je open source knihovna, která obsahuje soubor různých algoritmů pro zpracování úkolů z oblasti dolování z dat a nástroje na vizualizaci jejich výstupů. Knihovna je plně naimplementována v jazyce Java a je tedy multiplatformní. Vyvíjí se na University of Waikato již celých 20 let, přičemž Java implementace existuje od roku 1999.

Knihovna je rozdělena do tří hlavních částí (balíčků):

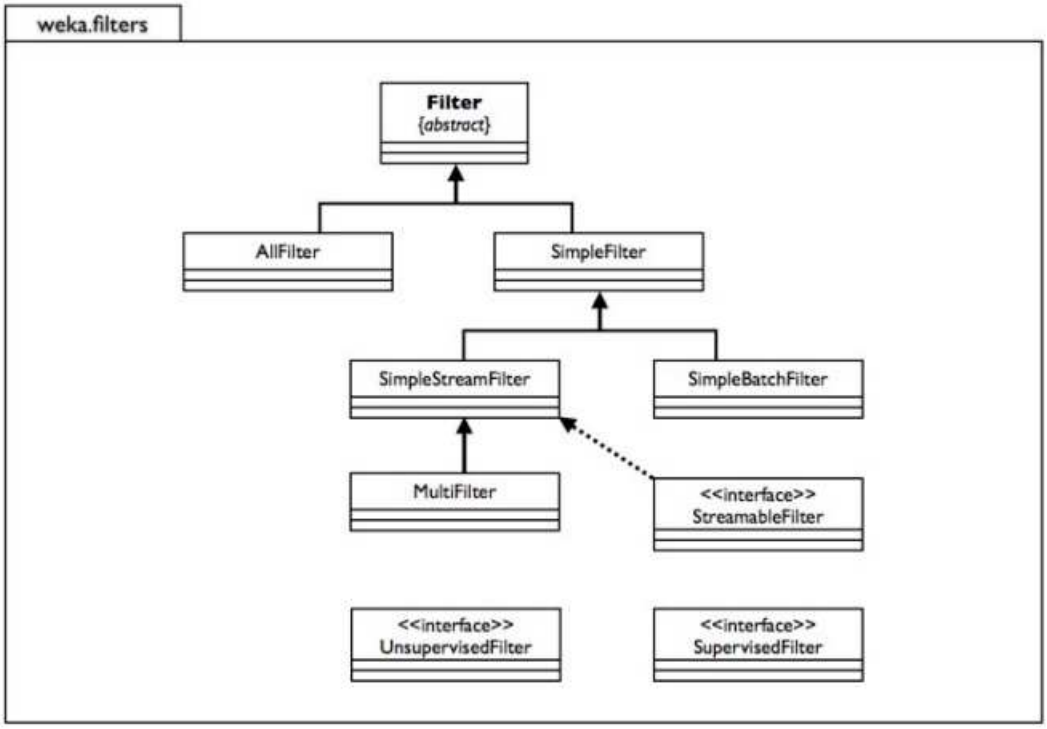
- weka.core
- weka.classifiers
- weka.filters



Obrázek 4-1 Převzato z [11]

Diagram zachycuje pouze zjednodušenou reprezentaci balíčku *core*. Především nám jde o typy *Instance* a *Instances*. Tyto dva typy jsou určeny pro uchovávání informací a dat, s kterými následně veškeré algoritmy pracují. *Instances* se používá jako datový soubor, pro uchování instancí jednotlivých atributů tabulky. Každý objekt typu *Instance* uchovává již vlastní data – vždy jeden řádek tabulky pro všechny atributy, spolu s daty také obsahuje reference zpět na atributy metadat. Všechny hodnoty jsou vnitřně uloženy jako čísla s desetinou čárkou – Double. Pokud je ukládaná hodnota řetězec, tak uložená hodnota představuje pouze index na korespondující nominální text v definici atributu.

Při implementaci operací pro transformaci v systému FIT-Miner budeme používat především filtry. Na následujícím diagramu je základní struktura, z které všechny filtry vycházejí. Pro spuštění jakéhokoliv filtru je zapotřebí použít statickou metodu *useFilter* z abstraktní třídy *Filter*.



Obrázek 4-2 Převzato z [11]

5 Analýza současného stavu

Současný stav je odrazem posledních změn, které na systému FIT-Miner byly provedeny v rámci diplomové práce Ing. Michala Šebka z roku 2009. V rámci ní byly provedeny změny v jádře, především také abstrakce a sjednocení přístupu k databázi.

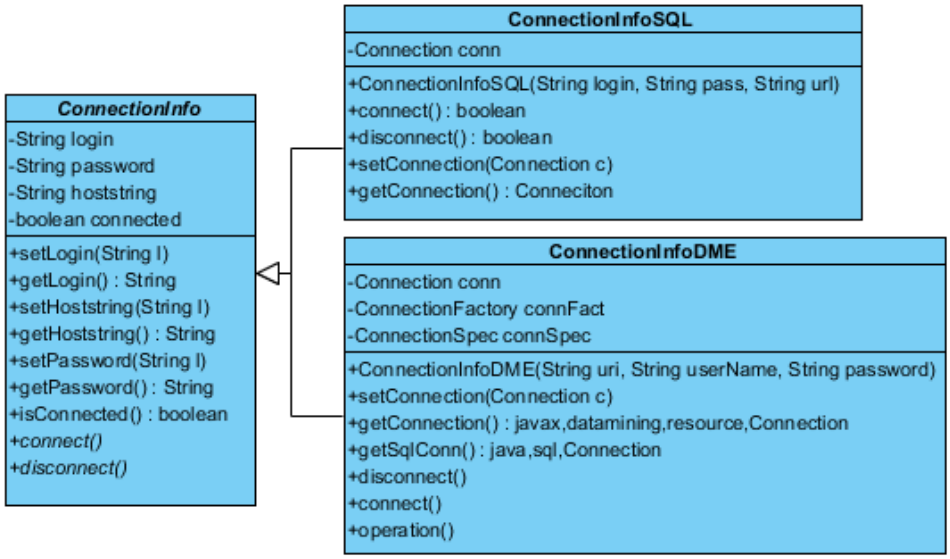
V této části se vynasnažím popsat veškeré části systému, které jsou nějakým způsobem spjaty s databází, tedy bude nutné do nich určitým způsobem zasahovat. Popíši zde i další části systému, které jsou důležité v rámci této diplomové práce.

5.1 Jádro systému

5.1.1 Připojení k databázi

Připojení k databázi je rozděleno na několik částí. Prvním krokem je získání parametrů pro vlastní připojení. Základní třídou pro uchovávání parametrů je *ConnectionInfo*. Z této abstraktní třídy v nynější podobě existují dvě konkrétní třídy.

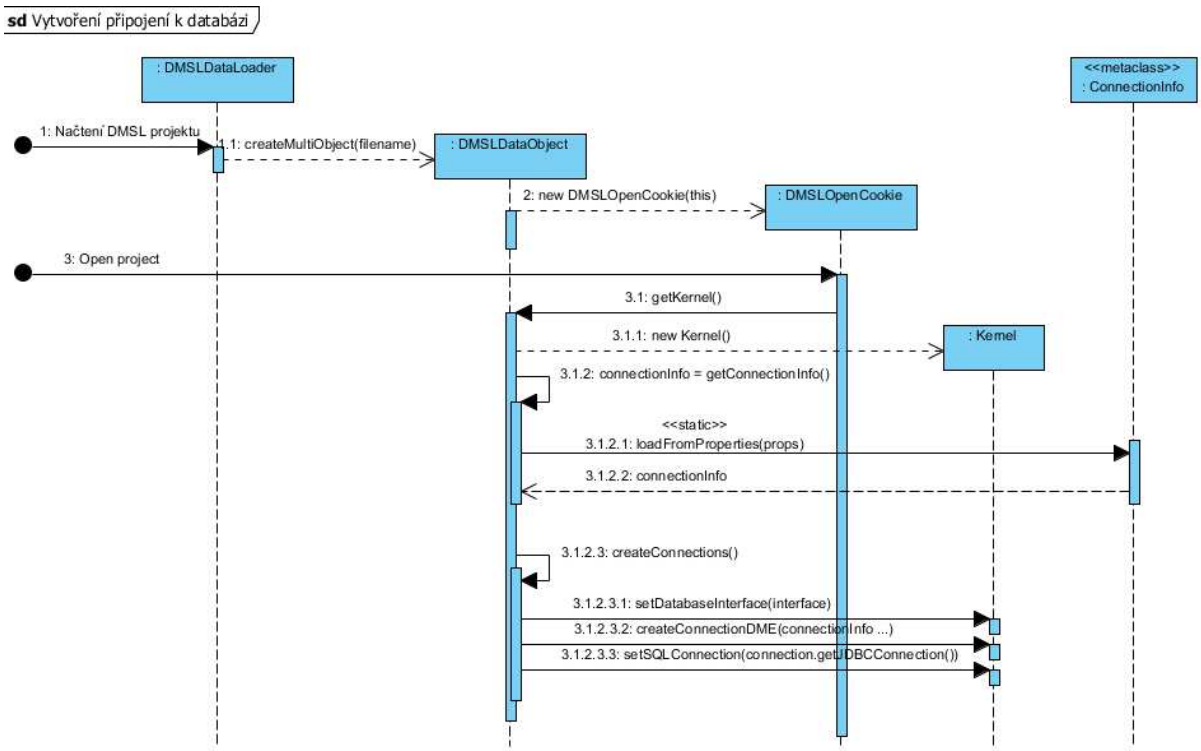
- *ConnectionInfoSQL* – používá se k standardnímu připojení k databázi. Třída obsahuje objekt *java.sql.Connection*, který uchovává aktivní připojení k databázi.
- *ConnectionInfoDME*, která se používá k připojení na Oracle Data Mining (ODM), dříve pojmenované jako Data Mining Engine, který poskytuje služby pro dolování z dat. Stejně jako standardní připojení do SQL si uchovává připojení, používá k tomu ale třídu *javax.datamining.resource.Connection*. Tato implementace je závislá přímo na Oracle databázi, a tudíž v rámci ostatních databázových prostředí nebude používána.



Obrázek 5-1 ConnectionInfo

V rámci třídy *ConnectionInfoSQL* jsou implementovány také operace *connect()* a *disconnect()*. Nicméně obě operace jsou označené jako zastaralé. Novější způsob počítá s tím, že připojování a odpojování je plně v režii jádra a *ConnectionInfoSQL* třída by do tohoto procesu tedy neměla zasahovat. Pouze by měla sloužit jako objekt pro uchování vlastního připojení a informací.

Obě připojení se inicializují při prvotním načítání *Kernel* objektu v rámci rozhraní *DMSLObject*, o kterém je řeč později, ovšem každé jiným způsobem. Po vytvoření *Kernel* objektu *DMSLObject* zavolá metodu *createConnections*, která se stará o inicializaci a připojení k DB. *ConnectionInfoDME* využije svoji metodu *connect()*, která je volána z *Kernel* objektu. *ConnectionInfoSQL* se inicializuje přímo v rámci *createConnections()* metody. Využívá k tomu interní *ConnectionManager* třídu z NetBeans platformy. Po inicializaci obou objektů je přístup k připojení již intuitivní přes jejich *Connection* objekty.



Obrázek 5-2 Sekvenční diagram inicializace projektu a vytvoření připojení

5.1.2 Ukládání parametrů připojení

V nynější verzi se ukládají parametry připojení v rámci jednotlivých projektů. Jsou ukládány do souboru project.properties umístěného v složce „<projekt>\dmproject“. Pro načítání atributů z properties souboru a jejich interpretaci existuje třída *cz.vutbr.fit.dataminer.api.ConnectionInfo*. Je to trochu matoucí název, jelikož se jedná o naprosto jinou třídu než tu, která existuje v rámci jádra a byla již zmíněna. Nicméně tato třída obsahuje definici názvů atributů ukládaných do souboru a statickou operaci *loadFromProperties(Properties properties)*, které z předaného *Properties* objektu, inicializovaného načtením všech atributů a jejich hodnot z vybraného souboru, vybere atributy pro připojení a vytvoří z nich novou instanci *cz.vutbr.fit.dataminer.api.ConnectionInfo*.
V nynějším stavu se ukládají následující atributy:

- connection.id – ID připojení podle kterého se dá najít v *ConnectionManager* třídě
- connection.name – Login/uživatel
- connection.url – JDBC string, definující URL připojení spolu s názvem databáze
- connection.passDME – Heslo do databáze, používá se, jak do DME, tak i do SQL připojení

5.1.3 Databázové rozhraní jádra IDatabaseAPI

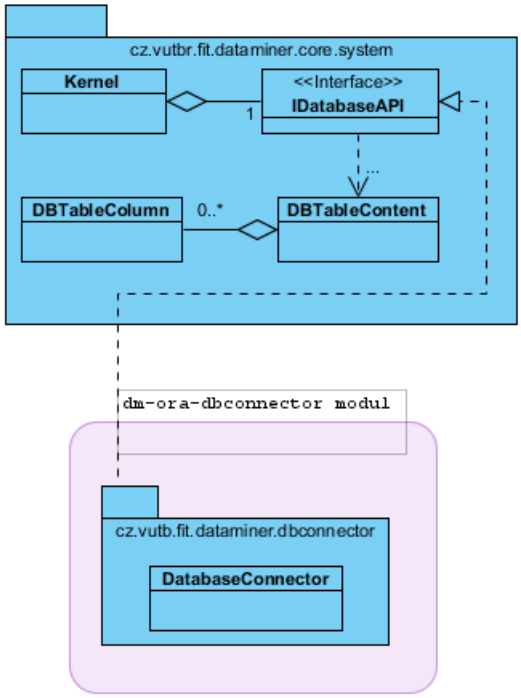
Toto rozhraní deklaruje veškeré operace potřebné pro práci s databází. Pro možnost používání různých SŘBD je důležité, aby veškerá komunikace v rámci jádra systému fungovala přes toto rozhraní. Jednotlivé moduly mohou samozřejmě přistupovat přímo k připojení, ovšem je nutné u nich ošetřit fungování na ostatních databázových systémech jednotlivě.

V rámci *IDatabaseAPI* rozhraní existují operace pro standardní práci s databází, jako například mazání, vytváření a dotazování nad tabulkami. Poté jsou zde také operace pro předzpracování dat. V neposlední řadě operace pro VIMEO funkce, které jsou v části předzpracování dat používány k čištění dat, doplnění chybějících hodnot a podobně.

Pro každý SŘBD, který má být použitelný pro FIT-Miner, musí být toto rozhraní implementováno. Prozatím je implementováno pouze pro Oracle v rámci třídy *DatabaseConnector*. *DatabaseConnector* třída definuje několik dalších operací a atributů, kromě těch, které existují v *IDatabaseAPI* rozhraní. V první řadě třída obsahuje oba objekty pro připojení k databázi – *ConnectionInfoDME, ConnectionInfoSQL*, které se jí předávají již v konstruktoru. Dalším atributem je DEBUG, který zajišťuje vypisování všech dotazů do databáze při vývoji aplikace a dalších užitečných informací. K důležitým operacím patří *db_loadColumns*. Tato operace načte do předpřipraveného objektu *DBTableContent*, za pomocí předaného SQL dotazu, všechna dostupná data. Diagram tříd rozhraní a jeho implementace je zobrazena na diagramu Obrázek 5-3.

Obrázek 5-3 Diagram tříd databázového rozhraní

Interface a vlastní implementace pracují s daty, která jsou uchovávána v rámci třídy *DBTableContent* a *DBTableColumn*, kam se načítají z databázového prostředí. Následující diagram Obrázek 5-4 ukazuje vazby nejdůležitějších tříd jádra zabývajících se prací s databází.



Obrázek 5-4 Diagram tříd relevantní k databázovému připojení

5.1.4 Třída Kernel

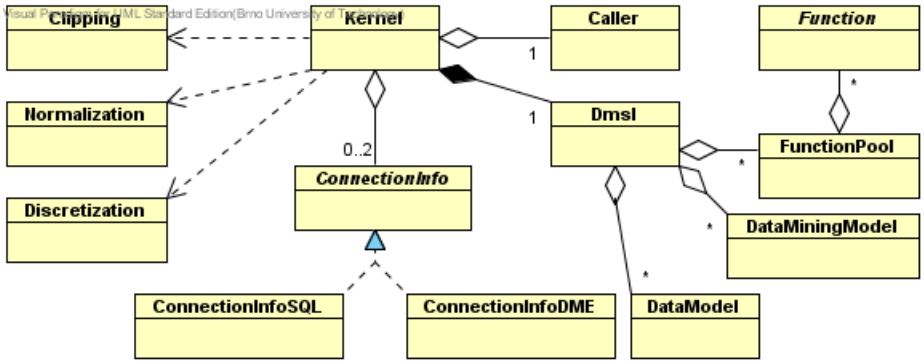
Při poslední úpravě od Ing. Michala Šebka byla tato třída změněna a neobsahuje tedy již tolik logiky. Veškerá práce s DMSL definicemi byla převedena do třídy *DMSL*. Kernel má v poslední verzi FIT-Miner systému několik funkcí.

Zejména uchovává jednotlivá připojení k databázi – *ConnectionInfoSQL* a *ConnectionInfoDME*. Také obsahuje atribut *databaseInterface*, který je nastaven na vytvořenou instanci implementace *IDatabaseAPI* rozhraní. V konečném důsledku tedy na *DatabaseConnector* objekt. Připojení se inicializují voláním operací *setSQLConnection(Connection)*, případně createConnectionDME(Connection) z třídy *DMSLDataObject* a to až po vytvoření vlastních spojení. Tyto dvě operace vytvoří nové *ConnectionInfo* objekty a pošlou je do databázového rozhraní, kde se taktéž uloží.

Druhou funkcí jsou operace pro ukládání nastavení DMSL souborů. Tyto operace jsou ovšem pouze prostředníky a volají vlastní implementaci v *DMSL* třídě.

V rámci *Kernel* třídy se taktéž řeší předzpracování dat. Důležitá je operace *processTransformations(String kernelBinding, String dataMiningModel)*, která provádí všechny požadované transformace na datech v logické posloupnosti a volá části databázové implementace. Stará se o vytvoření a smazání dočasných tabulek se vstupními daty pro jednotlivé transformace. Také vytváří finální tabulky pro naplnění transformovanými daty, respektive přejmenovává transformované tabulky na požadovaná jmena, aby se s nimi dalo dále pracovat.

Poslední částí, o kterou se *Kernel* stará, je aplikace VIMEO funkcí. Na to slouží operace *processVimeoFunctions(String kernelBinding, String dataMiningModel).* Třída *Kernel* a další základní třídy jádra jsou zobrazeny na diagramu Obrázek 5-5.



Obrázek 5-5 Diagram základních tříd jádra, převzato z [1]

5.1.5 Reprezentace databázových dat

O reprezentaci dat z databáze se v rámci jádra starají třídy *DBTableColumn* a *DBTableContent*.

DBTableContent obsahuje metadata o tabulce. Jde o její název, název databáze a také vektor, který obsahuje jednotlivé sloupce tabulky v podobě DBTableColumn objektů.

DBTableColumn, jak již název napovídá, obsahuje informace o jednom sloupci databázové tabulky. Obsahuje její jméno, typ, počet dat a také vektor, který v případě potřeby obsahuje všechny načtené data, nebo je prázdný, pokud nám stačí metadata o tabulce. Mimo to také samozřejmě obsahuje jméno tabulky a název databáze. Třída *DBTableColumn* obsahuje ještě operaci *setType(String dt)*, tato operace zajišťuje převod mezi databázovým názvem typu a typem používaným v rámci FIT-Miner systému.

Celkově je reprezentace dat poměrně dobře odstíněna od vlastní databázové implementace, jedinou výjimkou je právě operace pro převod typů, která bude muset být více generická.

5.1.6 Ostatní části jádra

5.1.6.1 Clipping a Normalizace

V rámci jádra systému existují třídy Clipping a Normalization. Obě jsou hodně podobné. Jejich účelem je koordinace volání jednotlivých typů těchto operací z databázového rozhraní, aby na sebe mohli navazovat. Jeden z jejich účelů je vytvářejí seznamy pomocí operace *makeExcludeLists()*. Každá operace má několik typů a každý výsledný sloupec může být zpracován pouze jedním typem. Tato operace tedy vytváří seznamy vynechaných sloupců pro jednotlivé typy, které se nebudou v rámci operace zpracovávat.

5.1.6.2 Discretization

Třída *Discretize* je podobně jako *Clipping* a *Normalization* třídy součástí předzpracování dat. Implementace se stará především o nastavení správných názvů dočasných tabulek, a kontroluje, zda sloupec, nad kterým se diskretizace bude spouštět, je určen pro diskretizaci v DMSL specifikaci.

5.1.6.3 ConnectionDial

Třída *ConnectionDial* má závislosti na databázovém připojení, nicméně nepodařilo se mi zjistit, že by se z nějakého zdroje používala. Jde spíše o starou třídu, která se nepoužívala. Tudíž byla v rámci implementace odstraněna a závislosti obsažené v ní se nemusí řešit.

5.2 DMSLDataObject a rozhraní DMSLObject

DMSLDataObject je implementací rozhraní *DMSLObject*. Rozhraní je definováno v rámci jádra systému, ale vlastní implementace existuje v modulu *dm-file-support*. Každý DMSLDataObject reprezentuje jeden projekt. Jedná se o třídu, která zajišťuje vytvoření *Kernel* objektu k jednotlivým DMSL specifikacím přes prvotní volání *getKernel()* operace. Při vytvoření *Kernel* objektu a získání připojovacích informací ze souboru se volá operace *createConnections()*. Tato operace kontroluje existenci Oracle driveru. Poté tento driver spolu s připojovacími údaji použije k připojení k databázi a to pro každé ze dvou připojení zvlášť. Nakonec do *Kernel* objektu, který v tuto chvíli již existuje, předá vytvořená připojení. Následuje vytvoření nové instance třídy *DatabaseConnector*, která se opět nastaví do *Kernel* objektu a zavolá na něj první příkazy pro nastavení databáze.

5.3 Grafické rozhraní

V rámci této diplomové práce je důležité z hlediska grafického rozhraní především vytváření nového projektu. To se děje v modulu dm-projects. Část, respektive jeden dialog, kde definuje vlastní připojení k databázi, existuje v šabloně ve třídě *EmptyDataMiningProjectPanelVisual2*. Je zde také část, která testuje připojení k databázi, zda je validní. Logika, která se stará o definování nastavení parametrů do nového projektu, se nachází v třídě *EmptyDataMiningProjectWizardIterator*.

5.4 Dolovací moduly

Velké množství modulů je i po posledních úpravách v jádru systému závislých přímo na Oracle implementaci dolování z dat. Zde jsou všechny vypsány, aby bylo jasné, jaké části jsou přímo závislé a budou tedy nutné upravit.

5.4.1 Dolování asociačních pravidel - ar modul

Tento modul je jednak závislý na použití *OraRulesFilter* filtru. Tento filtr ke své práci potřebuje několik dalších závislých tříd, které se musí provádět v rámci ODM, jedná se např. o třídy *AssociationSettings* nebo *AssociationModel*. Tyto a všechny další závislé nastavení fungují pouze na databázi, které podporuje JDM (Java Data Mining). Což je v současnosti pouze Oracle. Jedinou možností zprovozní je tedy využití Weka knihovny a aplikační logiky.

Dále se tu používají specifické hodnoty typů pro SQL dotaz, především *DM_Nested_Numerical*, který nebude kompatibilní s jinými databázovými systémy.

5.4.2 Detekce odlehlých hodnot - ad modul

Dolovací úloha pro detekci odlehlých hodnot je taktéž závislá na JDM, potažmo ODM. Prvně se zde provádí transformace v podobě normalizace hodnot v rámci *OraTransformationTask* třídy. Tato funkčnost bude nicméně v rámci *IDatabaseAPI*, takže by neměl být problém využít tu.

Modul dále používá *OraSVMClassificationSettings* třídu, a opět několik dalších nastavení, fungujících v rámci JDM. Kromě těchto závislostí se zde ještě nastavuje *nls_territory* atribut přímo do databáze, což je také specialita Oracle databáze.

5.4.3 Shlukování - ca modul

Tento modul je rozdělený na několik dolovacích algoritmů. Některé jsou plně závislé na Oracle implementaci, lze je poznat jednoduše již tak, že rozšiřují třídu *ODMClusteerTask*. Některé jsou vyřešeny již v rámci aplikační logiky, případě knihovny Weka a měli by být použitelné bez vážnějších úprav.

Algoritmy, které v nynějším stavu nejsou použitelné, zahrnují Enhanced K-means a Ocluster. Další, to znamená Denclue, DBscan a Optics jsou nezávislé na JDM.

5.4.4 Porovnávání výsledků shlukování - cc modul

Tento modul nemá žádné databázové závislosti. Ani v rámci Oracle, ani normálního SQL. Jediné závislosti spočívají v napojení na ca modul.

5.4.5 Klasifikace s využitím rozhodovacího stromu - dt modul

Stejně jako v modulu detekce odlehlých hodnot se zde používají závislosti na Oracle a JDM. Především pak nastavení klasifikace *ClassificationSettings*, *OraTreeSettings* a několik dalších. Z toho vyplívá, že modul používá databázový server pro dolování a není ho možné nativně použít s jinou implementací databáze. Také používá množství SQL příkazů, z čehož alespoň jeden je nekompatibilní s jinou databází, než Oracle a bude nutného ho předělat.

5.4.6 Genetický algoritmus – ga modul

Tento modul je pouze závislý na *OraConnection*, pro funkčnost modulu v jiných databázových systémech by mělo být dostačující vyměnit v inicializaci *GAModule* třídy DME připojení za SQL. Taktéž bude potřeba v třídě *DBFactory* upravit závislosti z *OraConnection* a *ConnectionInfoDME* na ty co implementují klasické připojení. Jelikož se ve všech případech DME připojení konvertuje zpět na klasické SQL a dotazy nejsou specifické pro Oracle, neměl by tu nastat žádný problém.

5.4.7 Víceúrovňové asociační pravidla – multi modul

Tento modul nemá žádné závislosti na Oracle. Nicméně používá své vlastní dotazy do databáze pro získávání metadat o tabulkách a sloupcích a pak také další dotazy, které je třeba prověřit na nových implementacích databází.

5.4.8 Bayesovská klasifikace - nb modul

Jednak zde existuje závislost na několika ODM tříd vztahujících se k implementované dolovací úloze, například *OraNaiveBayesModelDetail*. Dále jsou zde vytvořeny třídy *OraBinningTransformFactory* a *OraTransformationTaskFactory* které jsou určeny pro transformaci. Vlastní algoritmus je prováděn na straně serveru a je nutné ho opět převést, buď na algoritmus z Weka knihovny, nebo ho vytvořit od základu znovu.

Modul také provádí množství SQL dotazů přímo do databáze, nicméně všechny vypadají, že by neměly být problémové ani na jiné SŘBD.

5.4.9 Neuronová síť Backpropagation - nn modul

Tento modul je závislý znovu pouze na připojení *OraConneciton* a *ConnectionInfoDME* který ho obsahuje. Je tedy potřeba změnit ve třídách *BPDataLoad*, *Backporpagation* a *BackpropagationPane*l všechny přístupy do databáze na klasické SQL připojení a *ConnectionInfoSQL* třídu.

5.4.10 Prediktivní analýza - svm modul

Ze všech modulů je právě tento nejvíce závislý na specifické Oracle implementaci. Používají se zde třídy *OraNormalizeTransformImpl*, *OraNormalizeType* a *OraNormalizeTransformFactory*, které tvoří funkčně jednu skupinu. Dále také *OraTransformationTask* a O*raTransformationTaskFactory*,

Jedná se o třídy, jak již z názvu vyplývá, které se používají pro normalizaci jednotlivých položek v rámci předzpracování dat. Další dvě umí tyto normalizace provést na databázové vrstvě. Tyto závislosti by se měly dát zrušit jednoduše tak, že se použije nová implementace z *IDatabaseAPI* která již nebude na těchto třídách závislá.

Horší je to již se závislostí na vlastním zpracování regrese a to jak SVM tak GLM. Například třídy *OraGLMRegressionSettings*, *OraGLMSettingsFactory*, *OraGLMRegressionSettings*, *OraGLMSettingsFactory* a *OraGLMModelDetail*. Také veškeré tovární objekty, které jsou přes Oracle připojení vytvořeny, mají vlastní implementaci právě z Oracle tříd, jelikož žádná jiná neexistuje.

Jelikož přepsat celý modul tak, aby byl nezávislý na Oracle databázi, by bylo značně složité, nepředpokládám, že k tomu dojde v rámci této diplomové práce.

5.4.11 Predikce v časových řadách – es modul

Opět je nutné změnit veškeré výskyty Oracle připojení na klasické SQL. K tomu tento modul používá některé dotazy do databáze, které v sobě mají využívání funkcí, které nemusí být na ostatních databázových systémech k dispozici. Je tedy potřeba vyzkoušet, zda budou fungovat a případně je poté upravit. Také by bylo vhodné všechna volání posílat přes databázové rozhraní *IDatabaseAPI* a snažit se využít co nejvíce jeho potenciál.

6 Koncepce změn v systému

V této kapitole objasním koncepci změn systému, která vyplynula z provedené analýzy, tak jak je popsána v předchozí kapitole. Koncepce změn je rozdělena v rámci jednotlivých modulů a jádra systému, které bude ovšem mít na jednotlivé moduly také vliv a bude nutné je v reakci na změny v jádru měnit.

6.1 Celková koncepce

Celková koncepce byla pojata tak, aby se v budoucnu zajistila jednoduchá možnost přidávání dalších SŘBD. Úpravy v jádru systému směrovaly k větší univerzálnosti, především vytvoření nových rozhraní a továrních metod, které zajistily, že se při přidávání dalších SŘBD již nemusí upravovat zbytek jádra, či podpůrných modulů, kromě několika málo definic. V rámci dolovacích modulů bylo potřeba zajistit nezávislost a dodat možnost zjištění typu SŘBD.

6.2 Vybrané databázové systémy

Zde jsou ve stručnosti zmíněny databázové systémy, které budou v rámci této diplomové práce přidány a bude možné je použít pro práci se systémem FIT-Miner. Výběr byl proveden na základě analýzy různých SŘBD a po dohodě s vedoucím práce.

6.2.1 PostgreSQL

Jako první SŘBD pro zprovoznění na systému FIT-Miner byla vybrána PostgreSQL.

PostgreSQL je open source objektově-relační SŘBD, který je vyvíjen více než 15 let. Je multiplatformní, a jelikož se jedná o projekt vyvíjený komunitou lidí, existuje pro něj také spousta rozšíření. Poslední stabilní verze je 9.2.2. PostgreSQL má také podporu spousty vnitřních funkcí, které se mohou hodit při převádění specifické Oracle implementace na nové SŘBD.

K podpoře dolování z dat se bohužel u PostgreSQL nikde nic nepíše, tudíž předpokládám, že defaultně tuto podporu mít nebude a bude nutné používat databázi bez podpory JDM a dolování ze strany serveru. Půjde používat pouze přes standardní SQL připojení. Ovšem tím můžeme docílit toho, že některé moduly nebudou možné upravit do stavu, aby bez speciálního připojení na algoritmy pro dolování z dat fungovaly.

6.2.2 MySQL

Jelikož je MySQL jedna z nejrozšířenějších SŘBD, byla jako druhá vybrána právě tato. MySQL je multiplatformní relační SŘBD. V důsledku je MySQL nyní vlastněna také firmou Oracle a vyvíjena pod hlavičkou Sun Microsystems. Nynější verze MySQL, na které bude prováděno testování, je verze 5.7.

V rámci MySQL také není známa žádná podpora dolovacího modulu na straně serveru. Je tedy nutné opět používat pouze standardní SQL a zpracovávání nechat na aplikační logice.

6.3 Změny v jádru systému

Tato část pojednává o plánovaných změnách v jádru systému. Změny se týkají připojení k databázi, databázového rozhraní a několika dalších částí.

6.3.1 Databázové rozhraní IDatabaseAPI

Rozhraní, které spolupracuje s databází, je již poměrně dobře nezávisle definováno, není třeba proto jeho rozvržení příliš měnit. Nicméně v rámci implementace *DatabaseConnector* třídy, která slouží pro práci s SŘBD Oracle, existuje atribut *ConnectionInfoDME*, který je specifický pouze pro Oracle. Jeho implementace pak také obsahuje závislosti na Oracle knihovnách. Pro tuto závislost existují operace *db_setConnection* které tento atribut nastavují čímž je tato závislost zavlečena do všech implementací rozhraní *IDatabaseAPI*.

Vytvořilo se tedy nové rozhraní pro DME připojení, které má závislosti pouze na JDM specifikaci, pro případ, že by některý další SŘBD toto připojení požadoval. Poté se změnil atribut operace pro nastavení připojení z *ConnectionInfoDME* na toto nové rozhraní a bude se předávat objekt podle SŘBD. V Oracle implementaci se tento objekt přetypuje zpět na požadovaný typ *ConnectionInfoDME*. Jelikož žádný jiný SŘBD nejspíše nebude požadovat separátní připojení do databáze, bude toto připojení nastaveno ve většině případů na *null* a implementace pro tu či onu databázi se dle toho zařídí a nepoužije ho.

Celé rozhraní se samozřejmě znovu naimplementovalo pro obě nově podporované SŘBD. Většina operací tak, jak jsou implementovány pro Oracle, používá SQL/PLSQL a bylo tedy možné využít části stávajícího kódu i pro jiné SŘBD. Problém nastával při operacích diskretizace, ořezání a normalizace dat. Všechny tyto operace používají specifické transformace z Oracle SŘBD.

V rámci MySQL ani PostgreSQL tato funkcionalita neexistuje na databázové vrstvě. Nezbyla tedy jiná možnost, než naimplementovat logiku na aplikační vrstvě.

Také se vytvořila abstraktní třída, která implementuje některé z operací z *IDatabaseAPI* a obsahuje další pomocné atributy a operace. Část operací je obecná a dá se tedy využít napříč všemi implementacemi. Spousta operací na druhou stranu vyžadovala pouze minoritní změny, jako změna názvu tabulky s metadaty, změna typů a názvů speciálních sloupců.

Pro *IDatabaseAPI* se vytvořila třída pomocí návrhového vzoru *továrna*, která bude vracet správnou implementaci jako interface na základě nastavení vybraného SŘBD. Takto se zajistí možnost přidání nového SŘBD. Jednoduše se vytvoří nová implementace *IDatabaseAPI*, tato se přidá jako jedna z možností do factory třídy, a o to, aby vracela správnou implementaci, se již postará sama. Tato factory třída se používá v *DMSLDataObject* třídě při vytváření nových připojení a databázového rozhraní.

6.3.2 Kernel

Změny v třídě Kernel jsou poměrně malého rozsahu. Bylo potřeba pouze upravit operaci *setSQLConnection* tak, aby nastavovala do objektu *ConnectionInfoSQL* správné informace o databázovém typu a připojovacích údajích. Další, už pouze kosmetické změny, se týkají přesunu některých konstant, které se používají v implementaci *IDatabaseAPI*, jako je například název sloupce pro sloupec s ID jednotlivých řádků.

Vhodně se upravila i operace pro nastavování DME připojení createConnectionDME a to tak aby pracovala s rozhraním místo konkrétní implementací, tak jako to bude ve zbytku aplikace. A také aby reagovala na nastavení prázdného připojení. Taktéž se přidal atribut pro zjištění a nastavení typu

připojované databáze, aby mohli moduly lehce zjistit, zda budou s tou či onou databází fungovat či nikoliv.

6.3.3 ConnectionInfo

Pro *ConnectionInfoDME* se vytvoří nové rozhraní, které nebude závislé na Oracle implementaci ODM, ale pouze na JDM, tak, aby bylo možné ho někdy v budoucnu využít. Je proto nutné vytvořit i factory třídu pro vytváření instancí. Veškeré závislosti v rámci jádra i modulů bude potřeba změnit na nové rozhraní.

V třídě *ConnectionInfoSQL* se přidá nový parametr pro nastavení typu databáze. V rámci tohoto bude vytvořena také nová enumerace, která bude jednotlivé typy databází definovat, tak, aby bylo jednoduché další SŘBD přidat.

6.3.4 DBTableColumn

Typy, které se porovnávají a určují v rámci této třídy, je nutné mít možnost nadefinovat pro jiné SŘBD. Například varchar2, který normálně existuje v rámci Oracle databáze, již neexistuje v rámci MySQL. Tudíž bylo nutné tuto operaci extrahovat. Jednotlivé SŘBD mají nyní možnost definovat si vlastní typy ve svém modulu.

6.3.5 DMSLDataObject

DMSLDataObject obsahuje především pro nás důležitou operaci *createConnections*. Tato operace se upravila, jelikož obsahuje závislost na ovladači pro Oracle databázi. Také se zde nastavuje databázové rozhraní a DME připojení. Vlastní úprava spočívala ve zjištění databázového typu z *ConnectionInfoSQL* objektu na základě kterého se vyhledá patřičný ovladač.

Databázové rozhraní se nastavuje taktéž na základě předaného typu databáze

6.3.6 MiningPiece

Do *MiningPiece* třídy se přidal identifikátor oznamující pro které databáze je použitelná ta dolovací úloha, která bude odvozena. Nejlepší bylo použít seznam všech podporovaných databází, s tím, že defaultně bude nastavena podpora pouze pro Oracle. Všechny ostatní moduly si to budou moci ve své třídě změnit.

6.4 Změny v modulech

Jako první byly provedeny jednoduché změny, které umožnily kompatibilitu většiny modulů. Hlavní změna ve velkém množství modulů bylo zbavení se závislosti na *ConnectionDME*, které se někde objevuje i když není zapotřebí. Všude tam, kde to bylo možné, se nahradilo toto připojení na klasické SQL. Tím u několika modulů došlo k zprovoznění ve všech databázových implementacích.
Další změny by byli mnohem jednoduší, kdyby byla zajištěna podpora JDM některou databází, ovšem jak již bylo zmíněno v minulé kapitole, je to nepravděpodobné.

Znamená to, že pro každý modul uvedený v kapitole analýza, bude nutné naimplementovat poměrně složité algoritmy v rámci daného modulu. Díky čemuž ovšem nastává otázka rychlosti operací. Pokud se takovýto kód v Javě bude provádět nad velkým množstvím dat, může se stát, že

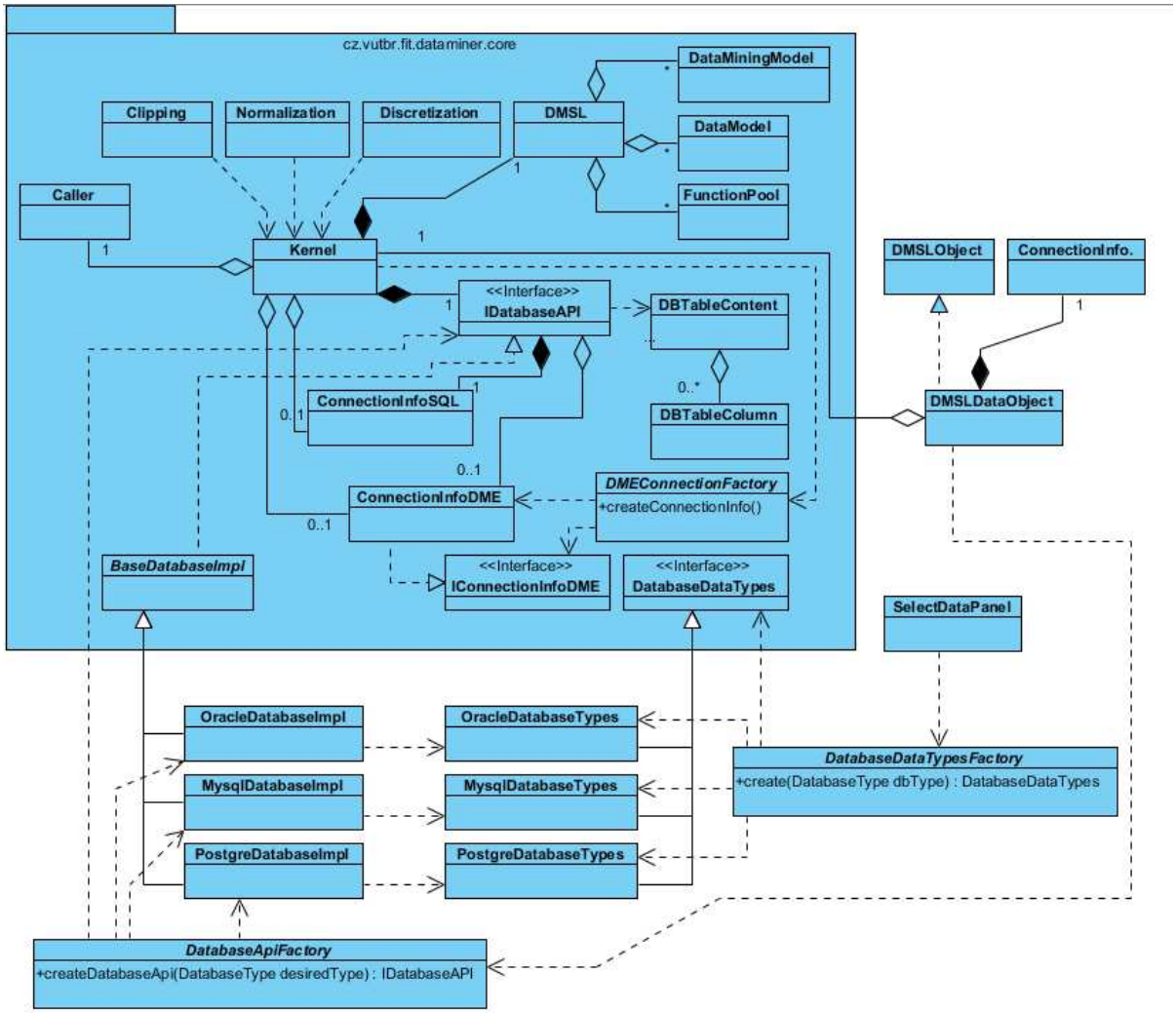
bude trvat několikrát déle, než ten samý kód volaný nativně v databázi. Taktéž paměťová náročnost může být neúměrně vysoká pro uživatele systému. Vždy ovšem bude záležet na konkrétním algoritmu a typu dolovací úlohy.
Pro všechny moduly bude nicméně platit velice podobné schéma, podle kterého se bude jejich struktura měnit. Pro ty třídy, které pracují přímo s ODM nebo JDM a bude zájem je předělat na nezávislé, se vytvoří rozhraní. Toto rozhraní bude definovat operace nutné pro dolovací algoritmus, které přicházejí do styku s JDM/ODM. V rámci modulu se vytvoří jedna třída, která bude představovat f*actory patern*. Tato třída pak bude vracet, na základě použité databáze ze třídy Kernel, implementaci pro daný SŘBD.

Dále již bude nutné jen pro každou databázi vytvořit novou implementaci. Pokud by se počítalo s přidáním SŘBD, která podporuje JDM standard, tak se vytvoří JDM implementace nebo upraví stávající Oracle a použije se pro více takovýchto SŘBD. Taktéž se může vytvořit defaultní implementace, která bude počítat s absencí vhodných metod v rámci dolování z dat a použije se na všech SŘBD, které podporu dolování z dat nemají.

Jednotlivé algoritmy, které budou potřeba řešit, jsou uvedeny z větší části v rámci kapitoly analýzy, ovšem je těžké dopředu odhadnout rozsah a požadovaný čas na jejich implementaci. Ty moduly, které byly plně závislé na Oracle implementaci, nebyly v rámci této diplomové práce zprovozněny.

7 Návrh a implementace

V této části jsou popsány reálné změny do FIT-Miner systému v rámci jádra a dalších modulů tak, jak byly naimplementovány. Implementace probíhala souběžně spolu s testováním funkcionality a porovnáváním výstupů vzhledem k původnímu kódu a výstupům z něho.



Obrázek 7-1Nové rozložení tříd jádra a pomocných modulů

Diagram na obrázku Obrázek 7-1 zobrazuje přehled převážně jádra a také nové tovární třídy *DatabaseAPIFactory*, *DatabaseDataTypesFactory* a *DMEConnectionFactory*, které se starají o správný výběr databáze. Dále jsou tu znázorněny závislosti mezi jednotlivými prvky jádra a to jak mezi novými tak starými.

7.1 Jádro systému a přidružené moduly

První částí je samozřejmě jádro systému a funkční moduly, které se nepodílejí na jednotlivých dolovacích úlohách. Zde byly nejdůležitější změny a největší část implementovaných operací. Především pak nové implementace *IDatabaseAPI* rozhraní pro dvě další SŘBD.

7.1.1 Identifikace typu databáze

Pro identifikaci typu databáze byla vytvořena enumerace *DatabaseType*. V nynějším stavu obsahuje typy ORACLE, POSTGRESQL a MYSQL. Každý typ enumerace kromě vlastního názvu obsahuje také název třídy ovladače dané databáze, který implementuje metody pro práci a připojení k databázi. Pak má atribut určující, zda smí používat separátní připojení pro JDM. Nyní to má povoleno pouze databáze ORACLE. Enumerace dále obsahuje jednu statickou operaci pro převod textového názvu databáze z konfiguračních souborů na *DatabaseType* entitu a druhou, která vrátí *DatabaseType* podle názvu třídy ovladače databáze. Defaultní *DatabaseType* je databáze ORACLE, aby se předešlo problémům při načítání projektů založených před možností používat více databázových systémů.

Dalším krokem byla úprava *ConnectionInfoSQL* třídy, do které se přidal atribut právě pro typ databáze jako nově vytvořená enumerace s defaultním nastavením na prázdnou hodnotu. Kvůli této změně se upravil i konstruktor aby počítal s novým atributem.

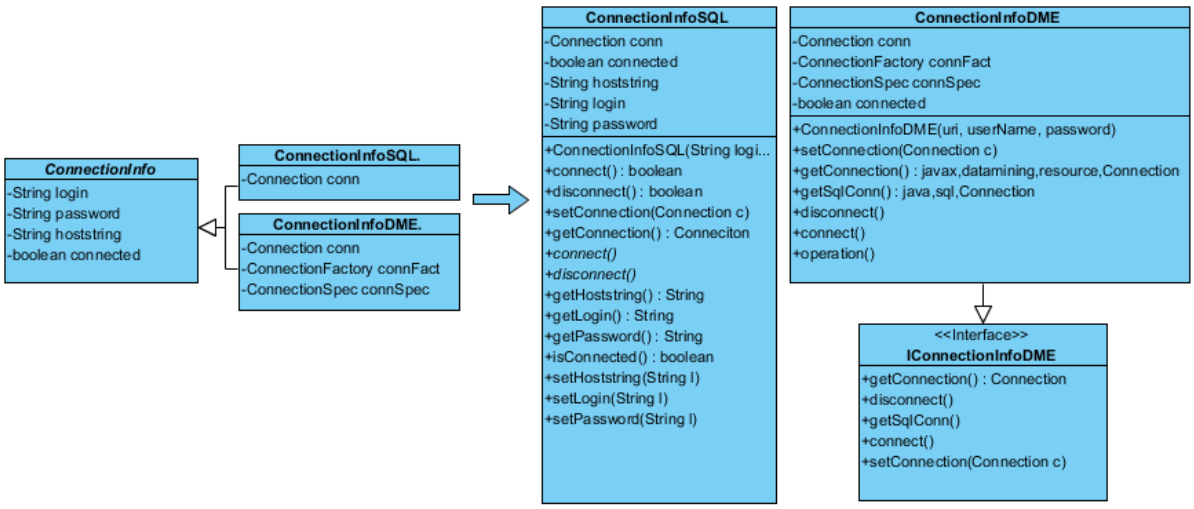
7.1.2 Výběr typu databáze

V rámci procesu vytváření nového projektu bylo potřeba přidat výběr SŘBD u dialogu databázového připojení. Vytvoření nového projektu se děje v modulu *dm-projects*. Výběr byl implementován jako combo box, jehož hodnoty jsou generované z *DatabaseType* enumerace. Hodnota typu databáze se také musela přidat do jako atribut do objektu *WizardDescriptor*, který se používá pro předávání parametrů v rámci dialogů při zakládání nového projektu.

7.1.3 Změny v ConnectionInfo třídách

Po analýze a naimplementování několika změn jsem nakonec došel k rozhodnutí, že není vhodné, aby obě třídy *ConnectionInfoSQL a ConnectionInfoDME* byly odvozeny ze stejné abstraktní třídy, jelikož prakticky nemají nic společného. Používají jinou třídu na udržení připojení i jiný systém pro držení vlastností tohoto připojení.

Proto jsem abstraktní třídu *ConnectionInfo* smazal a veškeré její atributy přenesl do *ConnectionInfoSQL*. Do *ConnectionInfoDME* bylo potřeba dodat pouze atribut, který určuje, zda je připojení aktivní. Jelikož ovšem třída *ConnectionInfoDME* obsahuje závislosti přímo na Oracle, vytvořil jsem ještě nové rozhraní *IConnectionInfoDME*, které definuje veškeré operace s použitím pouze JDM bez nutnosti závislostí na ODM. Toto rozhraní se použije jako náhrada za konkrétní implementaci v *IDatabaseAPI* a všude tam, kde se doteď pracuje přímo s *ConnectionInfoDME*. Také vznikla nová tovární třída *DMEConnectionFactory*, která vrací implementaci tohoto nového rozhraní – v současné době pouze Oracle, pro všechny ostatní databáze vrací null.



Obrázek 7-2Diagram tříd zobrazující změny v Connection třídách

7.1.4 Konverze databázových typů

Tato část se dotýká předně *DBTableColumn* třídy. Operace *setType(String dt)*, která pro jednotlivé sloupce určovala typ z *TYPES* třídy na základě textového názvu z databázového dotazu, byla smazána. Původní analýza počítala s tím, že se vytvoří abstraktní třída zaobalující veškerou funkcionalitu, kromě určování typů. Ty se poté měly implementovat ve zděděných třídách a takovéto třídy měly být vytvářeny přes návrhový vzor továrna. Tento přístup ale nakonec nebyl možný. Důvodem byly závislosti. Jednotlivé implementace databázového rozhraní jsou totiž ve vlastních modulech, a jak bylo již zmíněno, jádro nemůže být závislé na dalších modulech. To je výhodné, ale také to znamená, že v jádru systému nemůže existovat takováto konstrukce. Musel se nakonec zvolit jiný postup.

Jiný postup spočíval ve vytvoření rozhraní *DatabaseDataTypes*, které deklaruje pouze dvě metody – *convertTypesToDb(Types t)* a *convertDbToTypes(String dbtype)*. Jak už názvy napovídají, starají se o konverzi typů z databáze do *TYPES* a zpět. Následovalo vytvoření tovární třídy *DatabaseApiFactory* podle návrhového vzoru factory s jednou statickou operací. Tato vrací, podle předané enumerace *DatabaseType*, správnou implementaci *DatabaseDataTypes* rozhraní pro zvolenou databázi.

7.1.5 Kernel

V této třídě nastaly jen minoritní změny. Byla smazána operace *createSQLConnection*, jelikož již nebyla zapotřebí – používala se zřejmě dříve v rámci starého GUI. Dále byla upravena operace *setSQLConnection*. Jednak se nyní předává i typ databáze a veškeré atributy o připojení se nastavují i pro implementaci databázového rozhraní, což bylo do teď opomenuto a předávaly se nesmyslné informace – nejspíš do teď nebyly až tak potřeba, jelikož se využívalo vytvořené připojení přímo.
Taktéž byl změněn způsob vytváření *ConnectionInfoDME* objektu. Nyní se vytváří pomocí nové tovární třídy na základě typu databáze. Nakonec byl přidán atribut databaseType pro nastavení typu implementace databáze.

7.1.6 Vytváření vlastních připojení

První změna byla vyžadována v třídě *ConnectionInfo* z modulu dm-api, což je odlišná třída od již smazané třídy *ConnectionInfo* v jádru, nyní je tedy již jediná. Do třídy byl přidán atribut *databaseType* určující SŘBD z enumerace *DatabaseType*. Dále byla přidána konstanta definující název uloženého atributu v nastavení projektu – „connection.database“, která se nyní s každým uloženým projektem zaznamenává. Byla upravena i operace *loadFromProperties(Properties properties)*, tak aby vracela v rámci objektu i databázový typ, který je v případě nenalezení nastaven defaultně na ORACLE.

Vytváření připojení se zaobírá třída *DMSLDataObject*, přesněji operace *createConnections()*. Operace nejprve vybírá driver pro databázi. Proto bylo nutné změnit konstantu na jiný přístup, který název driveru zjistí z enumerace *DatabaseType*, podle zadané SŘBD ve změněném objektu *ConnectionInfo*. Následuje vytvoření připojení, které nebylo potřeba měnit. Správný driver se již o připojení postará. Další změna proběhla při vytváření objektu implementace databázového rozhraní pro *Kernel*. To se nyní nastavuje na objekt, který vyprodukuje třída *DatabaseApiFactory* přes statickou metodu *createDatabaseApi(DatabaseType type)*. Nastavení SQL připojení do *Kernel* objektu vyžadovalo také jen drobnou změnu – přidání parametru s typem databáze. Další změna byla v nastavení. Nastavení DME připojení je nyní vázáno na povolení v rámci enumerace databázového typu. Pokud databáze nemá příznak nastavený, vůbec se DME připojení neinicializuje a přeskočí se.

7.1.7 Databázové API

Interface *IDatabaseAPI* zůstalo stejné, jediná změna byla výměna *ConnectionInfoDME* za nový interface I*ConnectionInfoDME*, který není již závislý na Oracle implementaci.

Byla vytvořena abstraktní třída *BaseDatabaseImpl*, implementující část *IDatabaseAPI* rozhraní a sdružující podpůrné operace a atributy které jsou využívány všemi databázovými implementacemi. Třída uchovává klasické SQL připojení. Především obsahuje operace pro posílání dotazů do databáze a volání uložených procedur. Taktéž operaci pro načítání jednotlivých sloupců databáze a několik dalších. Umí nastavit parametr pro debuggování.

7.1.8 MiningPiece a zobrazování modulů

V rámci této třídy byly vytvořeny dvě nové operace *getAllowedDatabases()* a *isAllowedDatabase(DatabaseType type)*, která tu první přímo volá a kontroluje jí. Operaci *getAllowedDatabases()* musí každý dolovací modul přetížit a nastavit databáze sám, pokud chceme, aby byl použitelný na dalších SŘBD. Defaultně je nastaveno povolení pouze pro Oracle.

Spolu s touto změnou byla upravena i třída *MiningPaletteProvider* která se aktivně stará o vytváření palety dostupných dolovacích komponent tak, aby bylo možné je použít v rámci projektu. Do této třídy byl přidán odkaz na *Kernel* tak, aby mohl testovat jaká databáze je aktivní a mohl volat *isAllowedDatabase* na jednotlivých dolovacích modulech.

7.2 Weka wrapper modul

V projektu již existoval wrapper pro knihovnu Weka. Nicméně některé potřebné operace a třídy byly v modulech. Především načtení dat z *DBTableContent* do třídy *Instances*, tak aby data byla použitelná pro filtry a algoritmy. Proto jsem vytvořil balíček *weka.utils*, který je umístěn v modulu *dm-weka-wrapper*.

Balíček v současné době obsahuje pouze dvě třídy. Obě dvě byly přesunuty z původního modulu „dm-module-cc“. Jedná se o třídu *WekaDataset*, uchovávající vlastní *Instances* objekt a mapování mezi prvky tohoto objektu a primárním klíčem tabulky podle kterého lze data najít.

Druhou třídou je *WekaUtils*. Tato třída obsahuje především dvě statické metody pro načítání dat. Pro účely předzpracování byla metoda *loadInstacesFromDB* lehce upravena. Především bylo odebráno mapování mezi primárním klíčem a jednotlivými záznamy, jelikož tento primární klíč nebyl vždy k dispozici. Byl odmazán parametr pro primární klíč v signatuře metody. Taktéž jsem našel chybu v indexování pole při ukládání textových atributů, což způsobovalo vyhození výjimky.

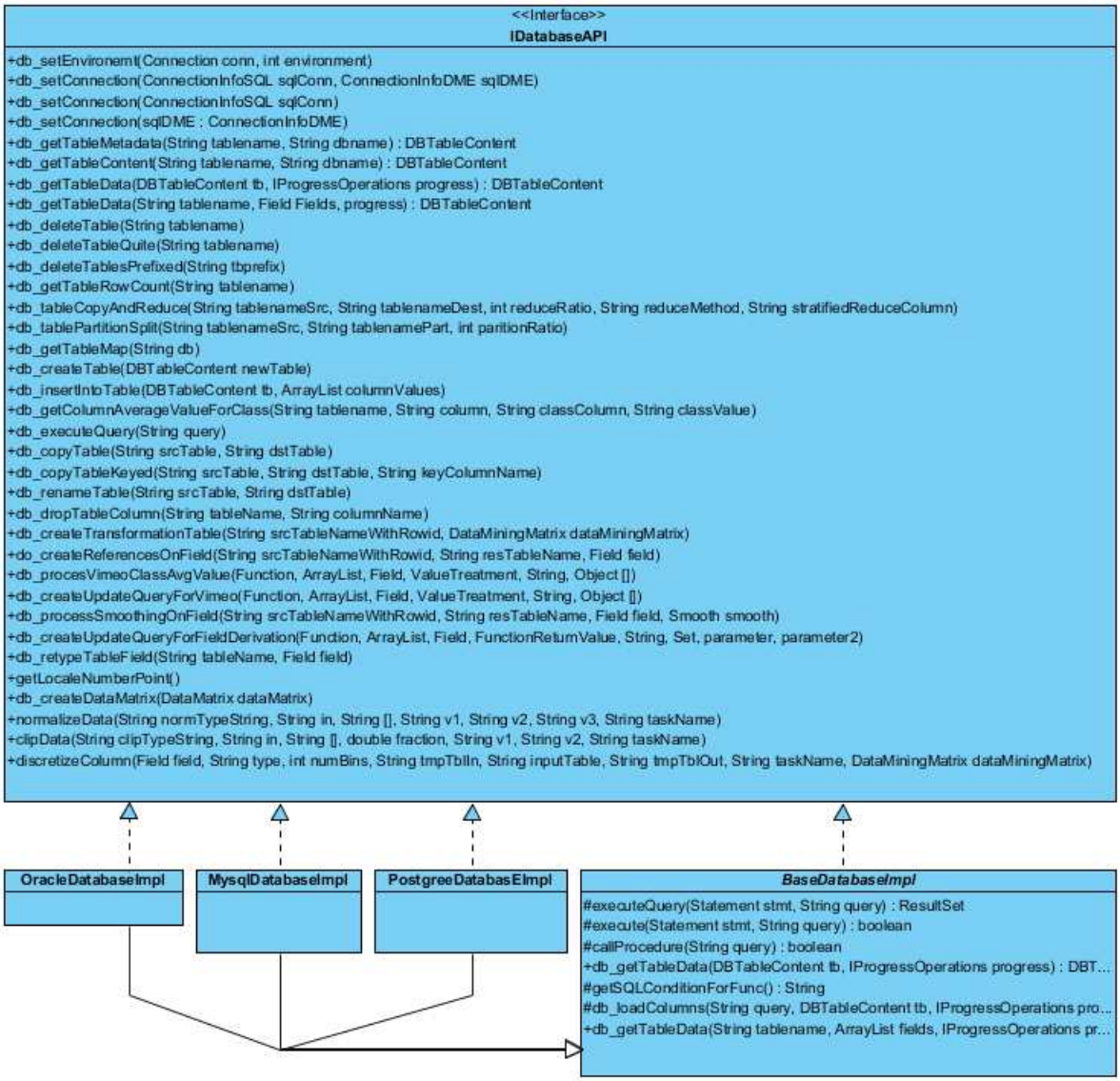
Poslední změnou pak bylo odstranění nominálních indexů pro celočíselný typ, jelikož při zpracovávání diskretizace se díky tomu špatně generovaly jednotlivé koše hodnot.

Původní metoda *loadInstacesFromDB*, beze změn, taktéž zůstala zachována pro potřeby dolovacího modulu.

7.3 Nové moduly pro databázové implementace

Pro dvě nové SŘBD, MySQL a PostgreSQL, bylo nutné vytvořit nové moduly. Jde o moduly **dm-postgree-dbconnector** a **dm-mysql-dbconnector**.

V rámci všech tří modulů existují dvě třídy. Jedna implementuje *IDatabaseAPI* rozhraní a druhá *DatabaseDataTypes*. Diagram Obrázek 7-3 zobrazuje nově vytvořené implementace rozhraní i novou abstraktní tříídu.



Obrázek 7-3 Diagram tříd nové implementace databáze

7.3.1 Oracle

Oracle již implementaci měl, ovšem aby byla zachována nějaká systematičnost v pojmenovávání, tak byla třída *DatabaseConnector* přejmenována na *OracleDatabaseImpl*. Taktéž byla tato třída přesunuta do balíčku *cz.vutbr.fit.dataminer.dbconnector.oracle*. Dále do stejného balíčku byla přidána nová třída *OracleDatabaseTypes*, implementující rozhraní *DatabaseDataTypes* pro převod datových typů mezi databází a FIT-Miner systémem.

V rámci vlastní implementace došlo pouze k menším změnám. Bylo přidáno dědění z abstraktní třídy *BaseDatabaseImpl* a tím pádem smazáno několik operací, které se nyní řeší v dané abstraktní třídě.

Vzhledem k úpravě *DBTableColumn* třídy, respektive odebrání funkcionality pro převod typů, bylo nutné u operací *db_getTableMap* a *db_getTableContentWithMetadata* využít pro určení typů třídu *OracleDatabaseTypes* a její operace.

Poslední změnou je úprava závislosti DME připojení, které se změnilo z *ConnectionInfoDME* na jeho nové rozhraní *IConnectionInfoDME*, nezávislé na Oracle implementaci.

7.3.2 PostgreSQL

Při vytváření nového modulu pro PostgreSQL již bylo potřeba zanalyzovat veškeré operace v rámci implementace pro Oracle a zjistit odlišnosti, které budou nutné upravit.

První krok byl vytvoření *PostgreDatabaseTypes* třídy a nadefinování správných typů pro PostgreSQL databázi, které se značně lišily od těch z databáze Oracle.

Následně se již vytvořila implementace *IDatabaseAPI* pro PostgreSQL – *PostgreDatabaseImpl*. Nyní následuje popis implementace jednotlivých důležitých operací. Pro druhou databázi již budou popsány jen odchylky od této implementace.

V rámci popisu jednotlivých operací jsou někdy vynechány určité parametry. Znamená to, že ten parametr se v této implementaci nepoužívá a je specifický pro SŘBD s podporou JDM, nebo je nedůležitý.

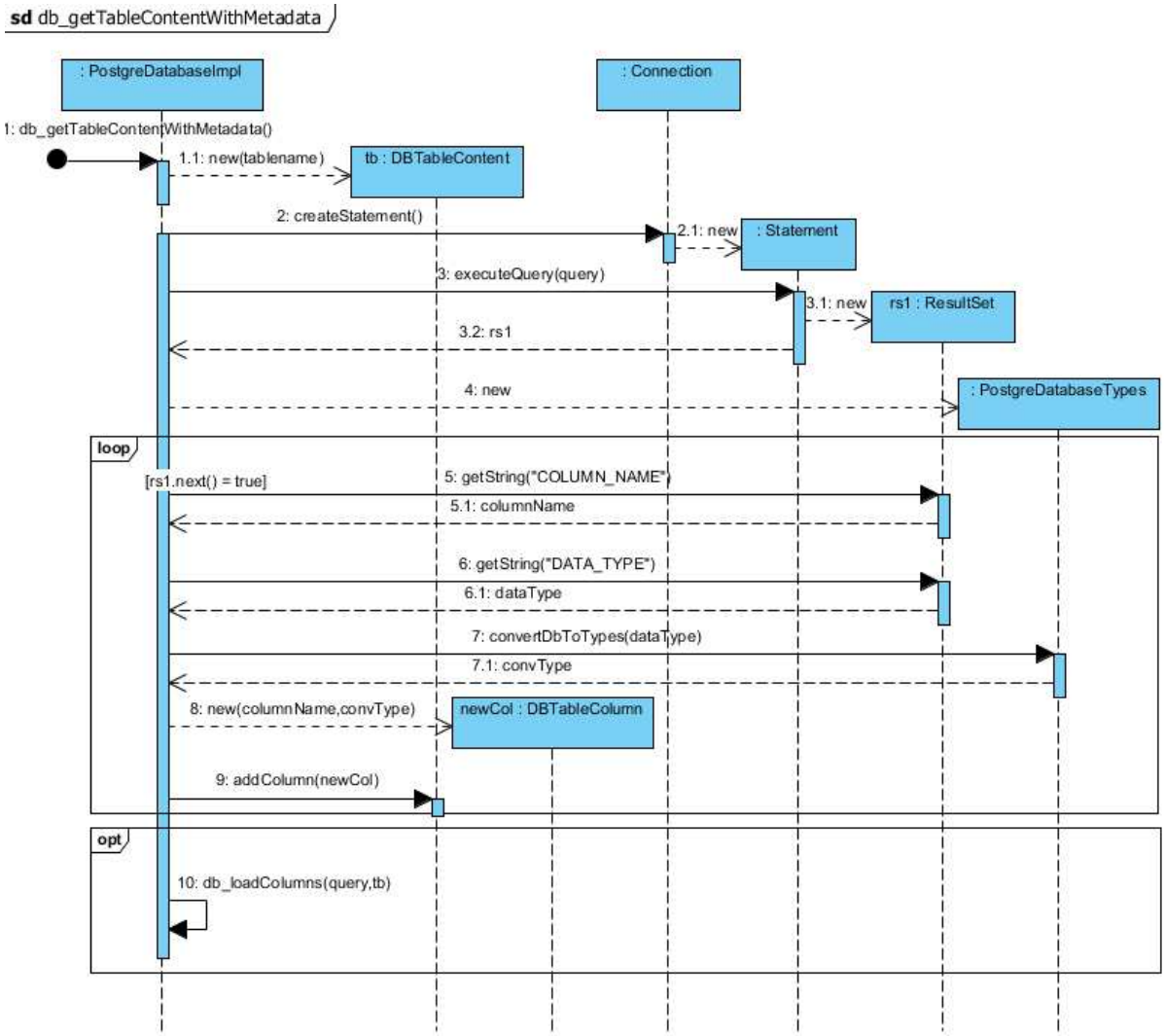
db_setConnection (IConnectionInfoDME cidme)

Tato operace byla implementována tak, aby při zavolání vyvolala výjimku pro nepodporovanou operaci, jelikož databáze speciální DME připojení nepodporuje.

DBTableContent db_getTableContentWithMetadata(String tablename, String dbname, boolean metadataOnly)

Jedná se o centrální operaci, která se stará o načítání metadat tabulek a případně i o volání operace, která načte vlastní data. Vstupními parametry jsou tablename, určující název tabulky. Dbname, který se ignoruje a metadataOnly, který určuje, zda se mají spolu s metadaty načítat i aktuální data či nikoliv.

Následující sekvenční diagram Obrázek 7-4 zobrazuje proces této operace.



Obrázek 7-4 Sekvenční diagram načítání metadat

Za zmínku v této operaci ještě stojí vlastní dotaz do databáze na informace o sloupcích vybrané tabulky. Tyto informace se v PostgreSQL zjišťují z databázové tabulky *information_schema.columns* kde se dá podle atributu *table_name* vyhledat jednotlivé sloupce zadané tabulky. Pro naše potřeby nám stačí získávat název a typ sloupců, který se poté převede do typu pro FIT-Miner.

Tato operace není definována v *IDatabaseAPI*, ale je volána z několika dalších operací které již v rozhraní jsou. Volá se s rozdílnými parametry – pro načtení pouze metadat, nebo i vlastních dat.

DBTableContent db_getTableData(DBTableContent tb, IProgressOperations progress)

První ze dvou operací, které mají stejný název a jsou tedy přetížené s jinými parametry. Tato první varianta byla přesunuta do abstraktní třídy *BaseDatabaseImpl* a obsahuje jako parametr objekt *DBTableContent*, do které pouze načte vlastní data zavoláním operace pro načítání dat *db_loadColumns*.

DBTableContent db_getTableData(String tablename, ArrayList<Field> fields, IProgressOperations progress)

Druhá varianta je již zajímavější. Tato metoda vytvoří *DBTableContent* který obsahuje všechny sloupce z parametru *fields*. Tyto sloupce mohou být jak ze zadané tabulky, tak umí nahrazovat i reference z kterékoliv jiné. Při naplňování *DBTableContent* objektu postupně vytváří SQL dotaz s JOIN klauzulemi. Nakonec volá *db_loadColumns* operaci s vygenerovaným dotazem. Tato operace se volá pouze z třídy *Insight* pro zobrazování dat.

void db_deleteTablesPrefixed(String tbprefix)

Operace pro smazání všech tabulek se zadaným prefixem. Vhodné při ukončování programu nebo zavírání DMSL projektu pro smazání dočasných tabulek.

Pro získání informací o názvech se využívá v PostgreSQL *information_schema.tables* tabulku, která obsahuje seznam všech tabulek. PostgreSQL taktéž všechna jména tabulek a sloupců nastavuje na malá písmena v případě, že nejsou v uvozovkách. Problém v tomto případě je, že dotaz používá klauzuli „LIKE“ která je následovaná prefixem, který musí být v uvozovkách. Bylo tedy nutné přidat kromě kontroly původního prefixu i kontrolu prefixu zkonvertovaného na malá písmena.

db_tablePartitionSplit(String tablenameSrc, String tablenamePart, int paritionRatio)

Tato operace zajišťuje vyjmutí částí dat z původní tabulky do druhé. Jedná se tedy o rozdělení dat mezi dvěma tabulkami podle parametru *partionRatio*, který určuje, kolik procent záznamů se přesune do nové tabulky.

Operace funguje tak, že se nejprve zkontroluje, zda neexistuje nová a temporální tabulka, případně se obě smažou. Vytvoří se temporální tabulka. Tady nastal první problém – PostgreSQL nemá přesný ekvivalent k rowid, který je specialitou Oracle databáze.

Původně jsem chtěl využít OID, jejichž generování je nicméně od verze PostgreSQL 8.1 defaultně vypnuté. Nakonec jsem využil systémový sloupec ctid, který sice není neměnný, ale mění se pouze při použití „VACUUM FULL“ SQL příkazu, což by nemělo představovat problém.

Další část spoléhala v Oracle implementaci na random generátor a rownum k vybrání určitého počtu náhodných řádků. V této implementaci jsem stejného efektu dosáhl využitím matematické funkce random(), která je v rámci PostgreSQL databáze k dispozici, a k omezení počtu řádků jsem využil klasickou klauzuli „LIMIT“.

Takto vybrané řádky se z temporální tabulky smažou a zůstanou v původní tabulce. Všechny ostatní se přesunou do nové tabulky v rámci jejího vytvoření. Následně se ty samé záznamy z původní tabulky smažou.

Map db_getTableMap(String db)

Zjišťuje všechny dostupné tabulky, jejich sloupce, typy a omezení velikosti pro textové typy. Požívá se například z třídy *Select* pro vybírání sloupců. Dotaz na databázi si nechá vrátit všechny sloupce z *information_schema.columns*, u kterých je i atribut tabulky ke které náleží. Pro každou novou tabulku je vytvořen záznam s právě zpracovávaným sloupcem. Pokud tabulka již ve výpisu existuje, sloupec se do ní pouze přidá. Pro zjištění maximální délky textových typů se zjišťuje na rozdíl od Oracle implementace atribut *character_maximum_length*. Jelikož PostgreSQL taktéž vrací informace o systémových sloupcích, tak jsem musel zakázat v dotazu schémata *information_schema* a *pg_catalog*, v kterých se tyto tabulky a sloupce nacházejí.

db_createTable (DBTableContent newTable)

Jednoduché vytváření nové tabulky podle názvu v objektu newTable. Zde stojí pouze za zmínku, že při vytváření tabulky používá k definici typu sloupců novou operaci *convertTypeToDb* z *PostgreDatabaseTypes* třídy.

db_copyTableKeyed(String srcTable, String dstTable, String keyColumnName)

Vytváří kopii původní tabulky s jedním sloupcem navíc, který identifikuje jednotlivé záznamy. Znovu se zde používá jako identifikátor „ctid“ sloupec, který se ukládá do názvu předávaný parametrem. Tento název byl přesunut z aplikační logiky, je nyní uložený v abstraktní třídě *BaseDatabaseImpl* jako konstanta *COLUMN_ROW_ID_NAME*, aby se v případě nutnosti mohl změnit pro všechny výskyty najednou.

db_processSmoothingOnField(String srcTableNameWithRowid, String resTableName, Field field, Smooth smooth)

Operace se stará o vyhlazení dat v rámci předzpracování. Parametr *srcTableNameWithRowid* udává vstupní tabulku a *resTableName* výstupní. *Field* je vlastní sloupec, který se má zpracovat a parametr *smooth* udává typ vyhlazení a obsahuje také počet košů neboli počet rozdělení hodnot.

Tato operace rozděluje podle nastaveného počtu košů hodnoty a nastavuje je v rámci daného koše buď na průměr, podle predikátu SQL dotazu, nebo na medián, který zjistí ze seřazené kolekce z databáze. To, zda jde o medián nebo o průměr, je nastaveno ve *Field* atributu. V rámci implementace bylo nutné zajistit čísla jednotlivých řádků, ale tak, aby byla závislá na řazení. Využila se konstrukce speciálně pro PostgreSQL - *row_number() OVER () AS row_num*. Tato konstrukce vytvoří v dotazu sloupec s čísly řádků, které jsou vždy jiné v závislosti na řazení.

db_createUpdateQueryForVimeo(Function function, ArrayList<FieldRef> argRefsArr, Field fieldToApply, ValueTreatment valueTreatment, String tableName, valueIntTypesPriorities)

Tato operace vytváří vlastní dotaz pro VIMEO funkce. Důležité vstupní parametry zahrnují název tabulky v *tableName* atributu, vlastní sloupec *fieldToApply* na který se VIMEO funkce aplikuje a *valueTreatment* hodnotu, která se použije.

PostgreSQL nedovoluje automatické konverze mezi typy, například jako Oracle, takže pří vkládání průměrné hodnoty, která byla s desetinou částí, do sloupce, který byl celočíselný, docházelo k výjimkám. Bylo tedy potřeba všude specificky definovat typ a případně zaokrouhlovat a převádět v rámci aplikace.

7.3.3 MySQL

V rámci MySQL již není potřeba popisovat znovu jednotlivé metody pro práci s databází. Zmíníme se pouze o několika změnách, které byli potřeba udělat vzhledem k implementaci PostgreSQL databázového rozhraní.

7.3.3.1 Databázové typy

MySQL má podobné typy jako PostgreSQL, nicméně má i některé odlišné, které musely být v rámci třídy *MysqlDatabaseTypes* nadefinovány. Taktéž má trochu odlišnou syntax při úpravě typů jednotlivých sloupců tabulky, takže bylo potřeba tyto dotazy upravit.

7.3.3.2 Identifikace řádků

Některé operace vyžadují identifikaci řádků pomocí *row_id*, tak aby byl unikátní. Bohužel v rámci MySQL žádný takový systémový sloupec neexistuje. Proto bylo nutné v každé z několika operací, které tuto vlastnost vyžadují, vytvořit sloupec nový a vygenerovat celou posloupnost ID pro tabulku.
K tomuto účelu existují dva přístupy.

První byl vytvořit nový sloupec jako typ *SERIAL*. Tento přístup automaticky zajistí vygenerování hodnot sloupce jako unikátní a není potřeba žádné další úpravy. Nicméně je tento sloupec určen jako primární klíč s příznakem *auto_increment*. Po prozkoumání všech možností jsem ale dospěl k závěru, že to ničemu nevadí. Tento příznak sice může být v rámci tabulky pouze jeden, jinak dochází ke konfliktu, ale při vytváření tabulky se nekopíruje, takže by se tam nikdy neměl další objevit.

Druhý způsob fungoval na principu nastavení relačních proměnných pro vytvoření nového sloupce s jednoznačným identifikátorem. Ovšem toto řešení bylo méně praktické, takže jsem ho zavrhl.

7.3.3.3 Schéma

MySQL využívá podobné schéma pro metadata jako PostgreSQL, nicméně názvy tabulek jsou lehce jiné. Především se zobrazovaly znovu systémové tabulky při dotazech na výběr zdrojových dat. To bylo ošetřeno zamezením hledání v systémových katalozích.

7.3.3.4 db_processSmoothingOnField

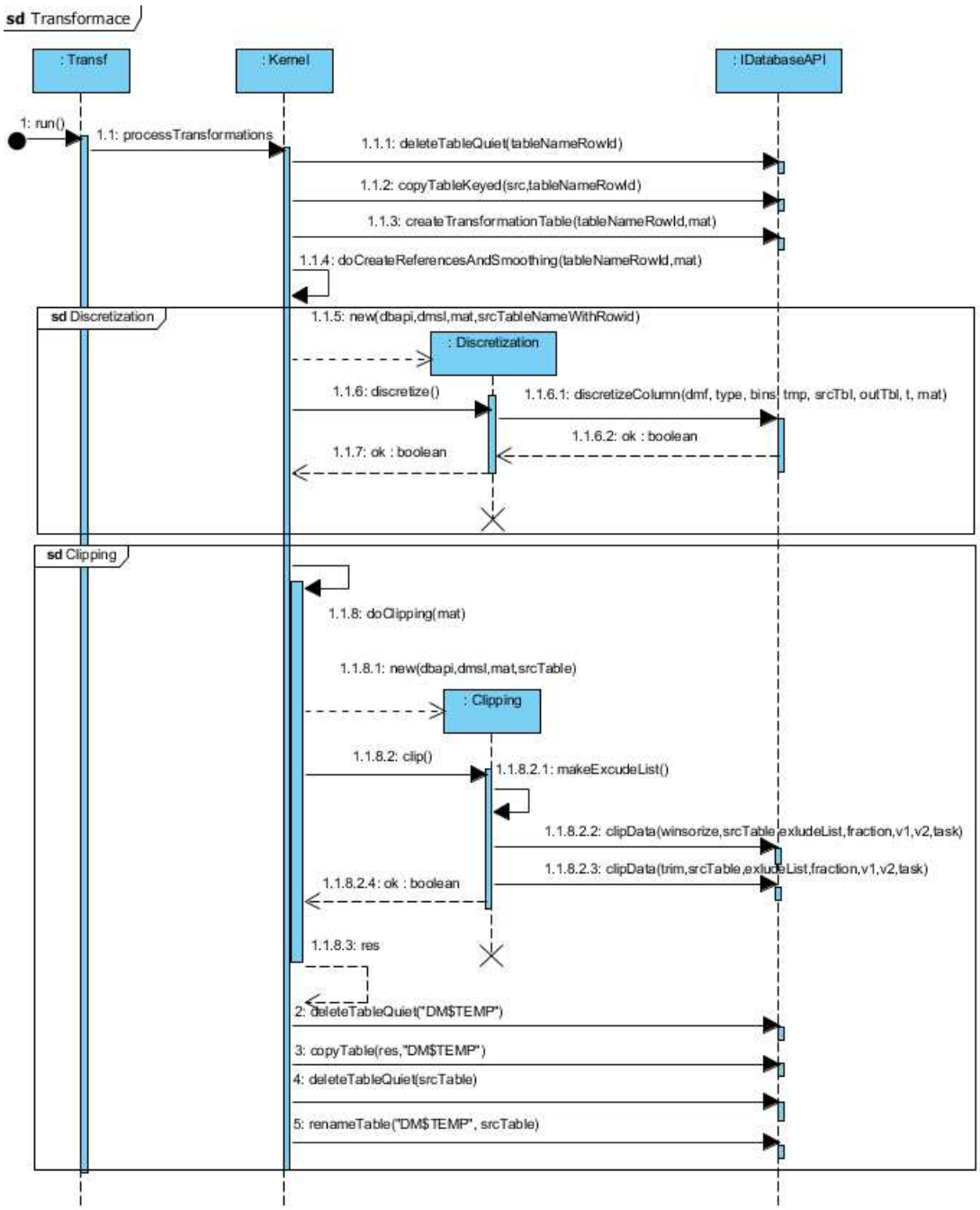
V rámci této metody bylo potřeba pouze pro SŘBD předělat vytváření generovaných dotazů. V PostgreSQL i v Oracle bylo možné používat čísla řádků, které se měnili spolu s řazením přes ORDER klauzuli. Toto nebylo v MySQL možné provést, takže se vytváření dotazů změnilo za pomoci LIMIT a OFFSET klauzulí, díky nimž bylo možné dosáhnout stejného výsledku s nutností pouze minimálního zásahu do logiky operace.

7.3.3.5 SQL uložené procedury

Pro PostgreSQL byli vytvořeny obě databázové procedury, jako má vytvořené Oracle. Pro MySQL byla vytvořena pouze první a po otestování nedoporučuji používat, jelikož tato implementace je pomalejší, než ta, která je v aplikaci. Je to způsobené trochu nestandartními požadavky na operace v rámci této uložené procedury, na které MySQL není úplně uzpůsobeno.

7.3.4 Transformace

Transformace se spouštějí z třídy *Transf*, poté se volá *Kernel*, který zajišťuje zpracování veškerých transformací, tak jak je uvedeno na následujícím grafu. Pro ušetření místa zde již není uvedena normalizace, nicméně se chová stejně jako clipping v diagramu Obrázek 7-5.



Obrázek 7-5 Průběh transformací

Prvně *Kernel* zajistí vytvoření nové tabulky s ID sloupcem a se všemi vstupními daty. Pak vytvoří výslednou tabulku pro transformaci s vybranými sloupci. Následně probíhá vyhlazení dat a vytvoření referencí na jednotlivé sloupce. Následuje vytvoření objektu *Discretization*, z kterého se následně

volá databázové rozhraní a provede se vlastní diskretizace nad všemi požadovanými sloupci. Po diskretizaci se vytvoří objekt *Clipping*, který nejdříve vytvoří list sloupců, pro něž se nemá provádět vlastní clipping a následně také volá databázové rozhraní pro vlastní operaci. Poté překopíruje výslednou tabulku do temp tabulky. Smaže původní výslednou tabulku a nahradí jí novou z právě provedeného clippingu. Takto se to opakuje poté i pro normalizaci.

7.3.4.1 Clipping

Clipping neboli ořezání dat bylo v původní databázové implementaci řešeno pomocí O*raClippingTransformFactory*, která samozřejmě má podporu jen v Oracle databázi. Proto bylo třeba vytvořit alternativní implementaci.

Pro ořezání jsem bohužel v rámci Weka knihovny nenašel žádnou ekvivalentní implementaci, která by se dala použít. Tudíž jsem funkčnost navrhl a vytvořil sám s pomocí SQL dotazů.

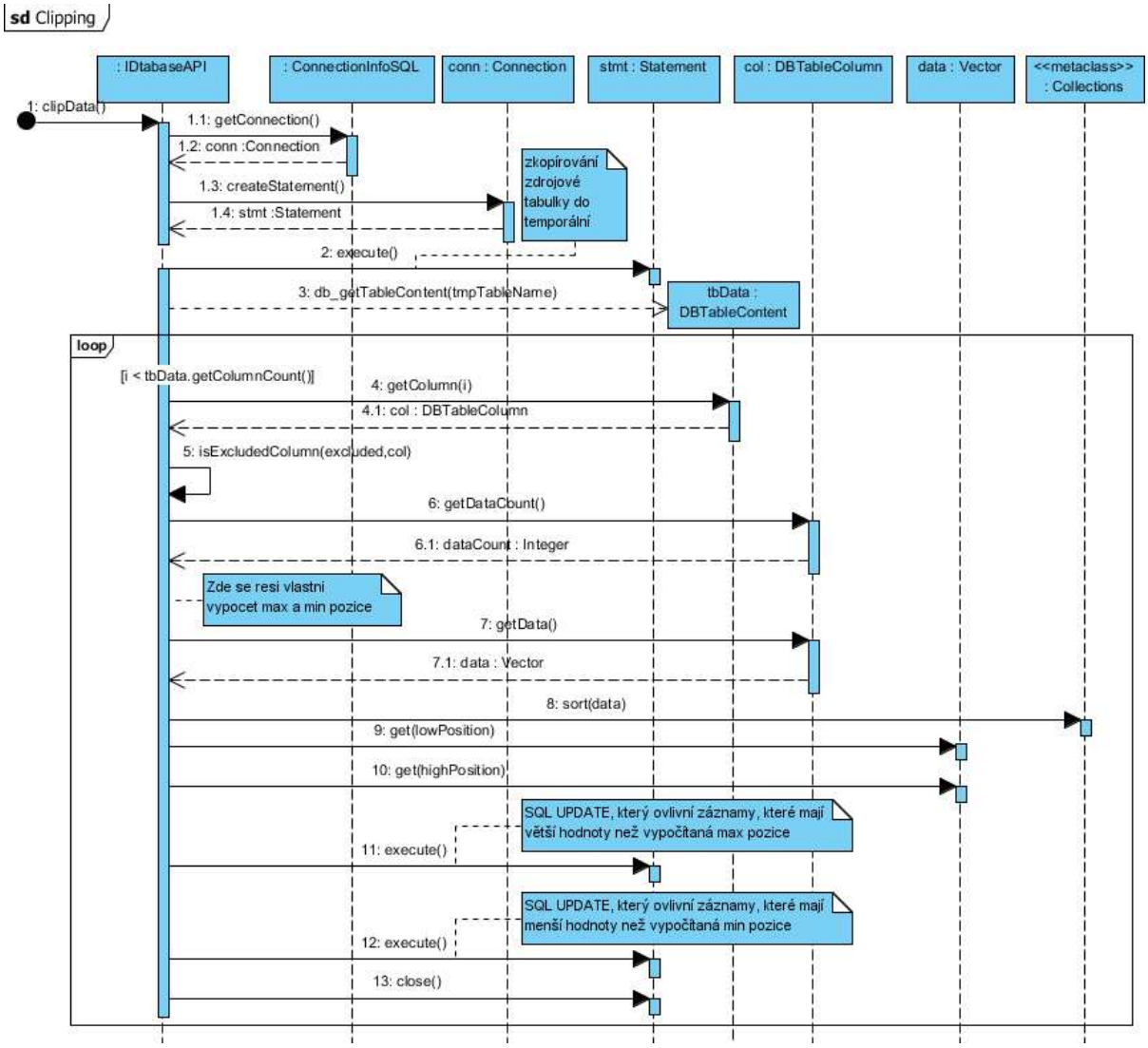
Signatura operace pro vlastní provedení ořezání v rámci implementace IDatabaseAPI je následující - ***clipData(String clipTypeString, String in, String[] excludedList, double fraction, String v1, String v2, String taskName)*.**

První parametr „*clipTypeString*“ určuje typ ořezání dat. Existují dva – winsorize a trim, které jsou definovány jako konstanty v třídě *Clipping* v balíčku DME v rámci jádra systému.

Druhý parametr „*in*“ je název vstupní tabulky. *ExcludedList* obsahuje seznam sloupců, nad kterými se ořezávání nebude provádět, a budou jednoduše zkopírovány.

Parametr *fraction* udává procentuální počet záznamů, které se z dat ořežou, a který může nabývat hodnot z intervalu (0,1). Poslední dva parametry jsou názvy výstupních tabulek. Třída *Clipping* je na základě typu ořezání nastavuje na název nebo null. Parametr *taskName* je specifický pro Oracle SŘBD a zde není důvod ho používat.

Na diagramu Obrázek 7-6 je sekvence provádění této operace.



Obrázek 7-6 Sekvenční diagram Clipping operace

Nejprve je vhodné jednoduše popsat dva již zmíněné typy ořezání, které lze použít. **Trim** jednoduše nahradí okrajové hodnoty prázdnou hodnotou null. **Winsorize** naproti tomu nastaví všechny okrajové hodnoty na poslední relevantní hodnotu, která zůstává na obou koncích nezměněna.

Clipping v této implementaci funguje na principu načtení všech dat. Postupného procházení jednotlivých sloupců a jejich zpracování. Zpracování v prvé řadě počítá se seřazenou posloupností. Tu jsem docílil použitím statické operace *Collections.sort(data)*, použitý na vektor všech dat zpracovávaného sloupce. Díky znalosti počtu záznamů a parametru *fraction* jsem mohl vypočítat pozici záznamu, který je hraniční. Bylo potřeba vybrat jeden záznam v horní a jeden ve spodní části záznamů.

Hodnoty těchto záznamů jsem využil při vytváření SQL UPDATE dotazů. Všechny hodnoty menší, případně větší byly podle typu ořezávání nastaveny na NULL nebo hodnotu hraničního záznamu.

7.2.4.3. Discretization

Taktéž jako ořezání byla diskretizace potřeba vyřešit přes jiné prostředky než JDM, potažmo Oracle ODM implementaci. Přičemž žádoucí bylo, aby implementace byla provedena se zachováním původních vstupů i výstupů algoritmu. V knihovně Weka existuje pro zpracování diskretizace filter *Discretize*. Existují dvě varianty. Jedna jakožto supervised filter, která se nejprve musí učit pomocí testovacích dat. A druhá, unsupervied verze, která funguje přímo s vstupními daty bez omezení. Pro potřebu FIT-Miner systému jsem vybral tedy druhou verzi. Signatura operace pro diskretizaci v IDatabaseAPI je následující:

discretizeColumn(Field field, String type, int numBins, String tmpTblIn, String inputTable, String tmpTblOut, String taskName, DataMiningMatrix dataMiningMatrix)

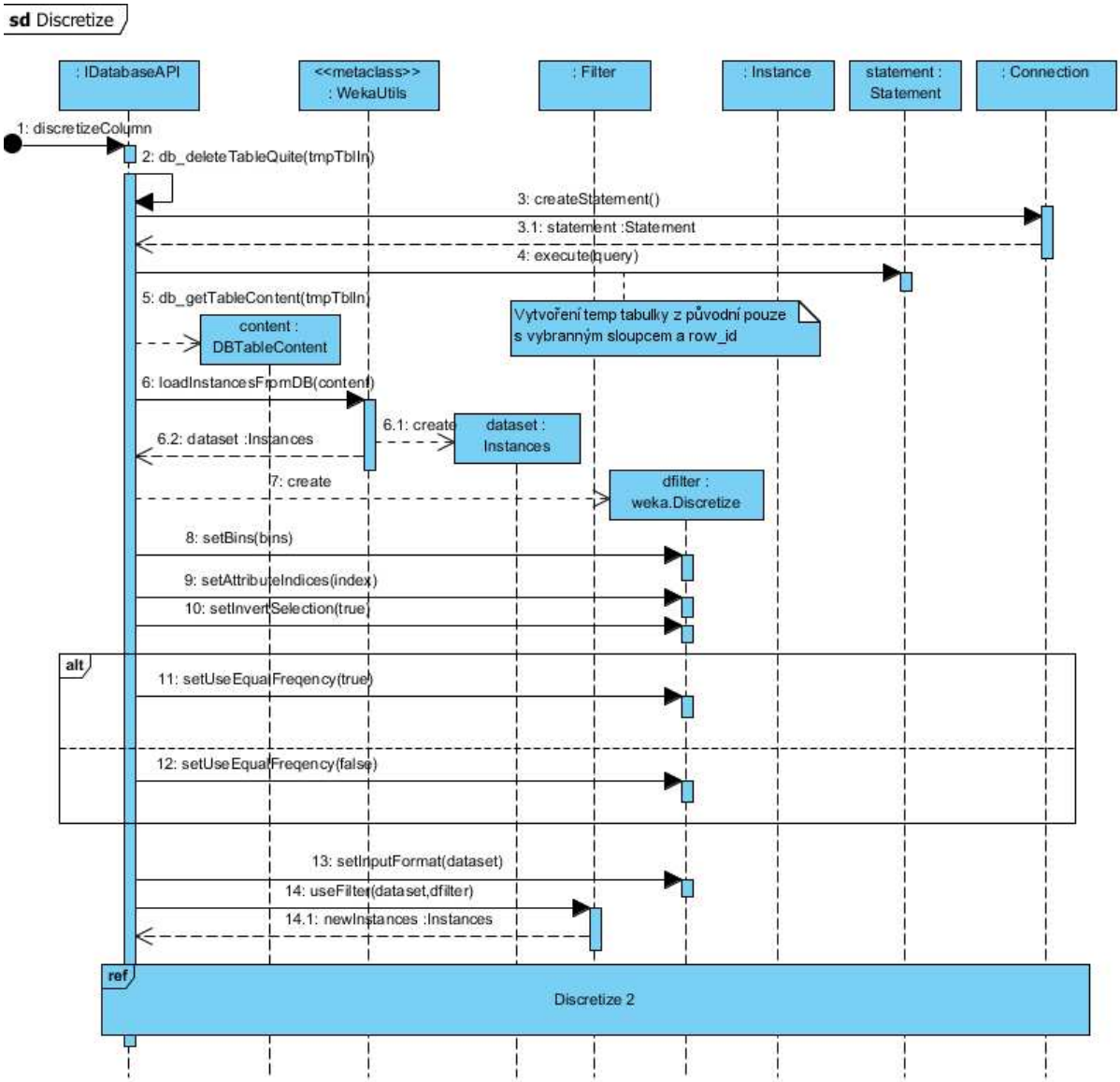
Parametr *field* obsahuje informace o sloupci, nad kterým se diskretizace bude provádět. *Type* parametr určuje, zda půjde o equal-width diskretizaci, což znamená rozčlenění na intervaly stejné šířky, nebo zda se použije quantile diskretizace, která naopak rozčlenění podle stejné hloubky/frekvence. Třetí důležitý parametr je numBins, který určuje počet „košů“ neboli intervalů, do kterých se hodnoty rozčlení. Další parametry *tmbTblIn* a *inputTable* pouze předávají jména tabulek, vstupní s daty a jméno temporální. Temporální se na konci operace smaže. TmpTblOut a taskName se v této implementaci nepoužívají a jsou validní pouze pro Oracle implementaci. Poslední parametr *dataMiningMatrix* obsahuje informace o původní vstupní tabulce, která bude po skončení operace zachována a bude obsahovat transformovaný sloupec.

Sekvenční diagram na obrázku Obrázek 7-7 zobrazuje první část diskretizace. Provádí se načtení dat a metadata o tabulce z databáze, které se uloží do třídy *DBTableContent*.

Poté následuje převedení načtených dat do typu, který je srozumitelný pro knihovnu Weka, čímž je typ *Instances*, popřípadě *Instance*. Pro tento převod byla použita operace *loadInstanceFromDB* v rámci *WekaUtils* modulu. Funkce bere jako parametr *DBTableContent*.
Vytvoří se filter *Discretize* a nastaví se mu všechny požadované vlastnosti – typ diskretizace, počet „košů“ a vlastní data, podle kterých filter rozpozná jejich strukturu.

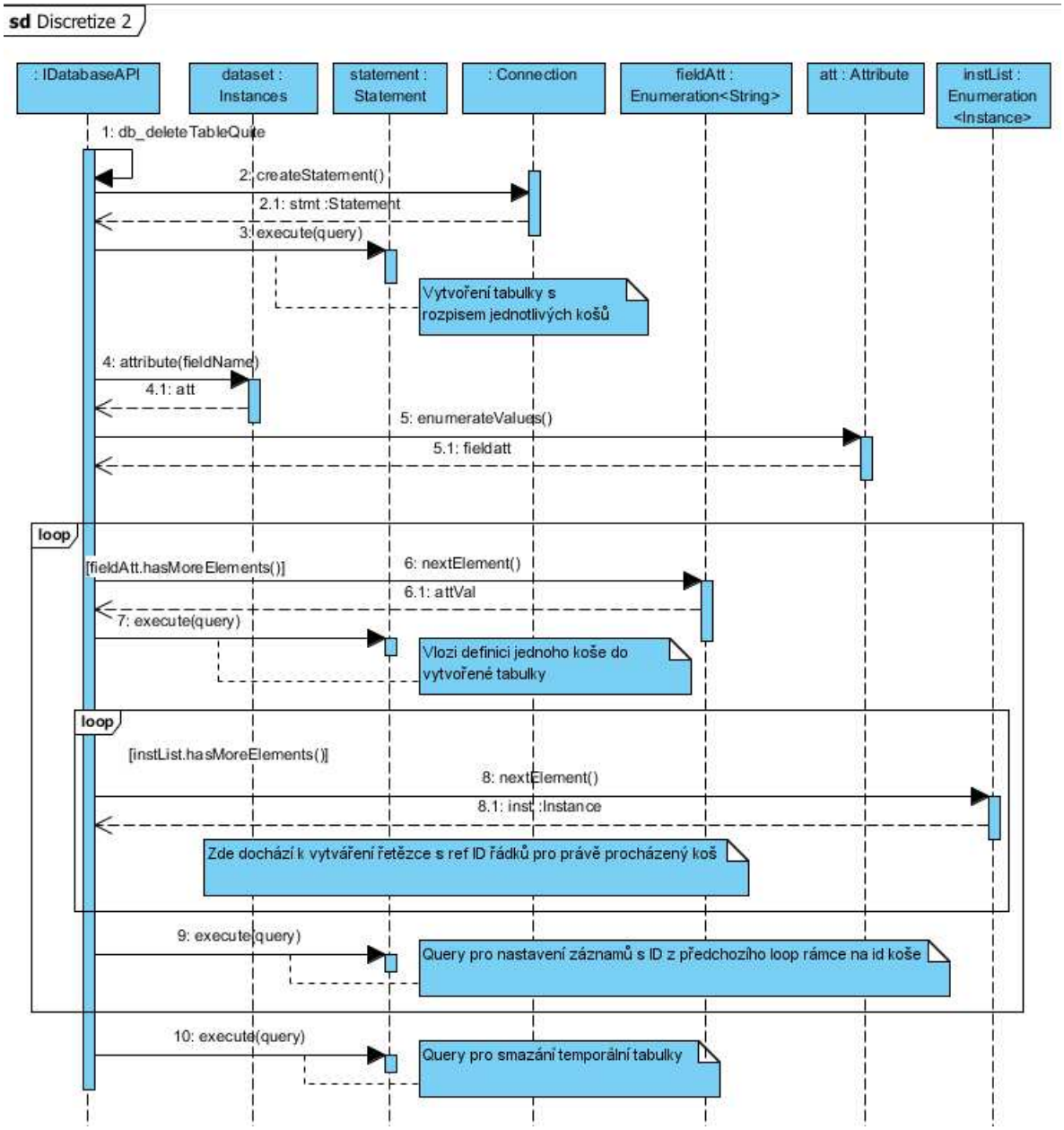
Také je nutné nastavit sloupce, které se mají při diskretizaci ignorovat. Tímto sloupcem je „*row_id*“. V datasetu si najdeme jeho index a použijeme ho jako parametr do metody *setAttributeIndicies*, který se používá na identifikování řádku v původní tabulce.

Nakonec se provede vlastní operace filtrování přes statickou oepraci *Filter.useFilter*.



Obrázek 7-7 Sekvenční diagram první části diskretizace sloupce

V tuto chvíli máme nové data a je potřeba je z formátu Weka knihovny převézt zpět do databáze. Další postup zachycuje následující sekvenční diagram na obrázku Obrázek 7-8.



Obrázek 7-8 Druhá část diskretizace - nahrávání dat zpět do DB

Pro vložení dat zpět do databáze je nutné vytvořit nejprve tabulku, která obsahuje ID jednotlivých košů spolu s popisem rozmezí intervalů, které koši náleží. Po vytvoření této tabulky se načte seznam košů z výsledného *Instances* objektu. Následuje cyklus, který všechny koše zpracuje. Nejprve se vloží definice koše a id do referenční, právě vytvořené tabulky. Následně se projde seznam všech hodnot diskretizovaného sloupce v *Instances* objektu. Ty hodnoty, které náleží do právě zpracovávaného koše, jsou vybrány a jejich identifikátor je vložen do textového řetězce ve tvaru *„id, id1, id2“* pro všechny id které vyhovují. Řetězec je na konci zpracování každého koše vložen do predikátu IN v sekci WHERE SQL dotazu.

Tento dotaz zajišťuje změnu hodnot diskretizovaného sloupce v cílové tabulce na ID koše tak, aby se nastavili pouze řádky, které tomuto koši mají náležet.

Nakonec se smaže temporální tabulka a vrátí se hodnota true, indikující úspěšné provedení diskretizace.

7.3.4.2 Normalization

Operace normalizace v databázovém rozhraní má následující signaturu

normalizeData(String normTypeString, String in, String[] excludedList, String v1, String v2, String v3, String taskName)

Parametr *in* určuje název tabulky se vstupními daty. *ExludedList* opět obsahuje sloupce, které se nemají normalizovat. V1, v2 a v3 jsou názvy tabulek pro uložení výsledných dat. Nejdůležitějším parametrem je tu normTypeString, který určuje, jaký ze tří typů normalizace se použije.

Všechny tři druhy normalizace nejprve byli naimplementováni pomocí dolovacích algoritmů z Weka knihovny. Ovšem tento přístup je znatelně pomalejší, než zpracování na serveru, především z důvodu nutnosti posílat pro každý záznam SQL dotaz na nastavení nové, normalizované hodnoty. Z tohoto důvodu bylo pro všechny algoritmy navrhnuto nové řešení přímo přes SQL dotazy a agregátní funkce, které nechá zpracování právě na straně serveru.

Pro všechny tři typy normalizace se jako první vybere správný název výsledné tabulky. Poté se vygeneruje seznam sloupců pro zpracování, s tím, že se zohlední speciální sloupec spolu s těmi, které jsou uvedeny jako nechtěné. Poslední sdílenou položkou je vytvoření temporální tabulky, která obsahuje data pro vlastní normalizaci.

Min-Max normalizace

První typ normalizace, která byl implementován, je Min-Max. Algoritmus postupně projíždí sloupce a zpracovává je po jednom.

První implementace přes algoritmus knihovny Weka fungoval na bázi načtení dat, podobným způsobem jako u diskretizace. Zpracoval je přes *Filter* třídu a poté je znovu zařazoval postupně databáze, pro každý záznam bylo nutné vytvořit vlastní SQL dotaz.

Pro druhou, rychlejší verzi je nejprve nutné sloupec přetypovat na typ s plovoucí desetinou čárkou, aby se nad ním mohly provádět další operace a ukládat výsledek v požadovaném tvaru. Poté se z databáze zjistí minimum a maximum daného sloupce přes agregační funkce. Jelikož u Min-Max normalizace chceme rozsah hodnot přizpůsobit intervalu <0,1> tak se tyto hodnoty použijí v rámci následující vzorce.

$$v' = \frac{v - min}{max - min}(nmax - nmin)$$

Pro update všech hodnot najednou, lze použít SQL Update příkaz, bez použití WHERE části s odkazem na všechna data ze sloupce. Hodnoty jednotlivých řádků lze získat použitím názvu sloupce. Tím pádem lze vytvořit předpis pro všechny řádky a nastavit hodnoty v rámci jediného dotazu.

V rámci PostgreSQL ještě nastával problém v zaokrouhlování a někdy se stalo, že hodnoty, které měly být nulové, nebyly. Proto v rámci implementace na PostgreSQL následuje druhý dotaz, který všechny hodnoty, které jsou menší, než hodnota 0.0000000001 transformuje na 0.

Lineární normalizace

Druhý typ normalizace se chová podobě jako první. Opět byl nejprve implementován pomocí knihovny Weka. Jelikož to bylo ovšem pomalé, přešlo se na stejný jednoduchý UPDATE model, jako u Min-Max normalizace. Jediný rozdíl je, že u této normalizace použijeme jiný vzorec pro update všech hodnot. Obě hodnoty jsou převedeny na absolutní hodnotu a větší z nich je pak použita k vydělení každého prvku v datech tohoto sloupce.

Z-Score normalizace

Pro tento typ normalizace má Weka filter nazvány *Standardize*. Jelikož se nejprve použila tato implementace, tak je zde také popsána.

Prvním krokem k použití tohoto filtru je vytvoření další pomocné tabulky, která obsahuje pouze ty sloupce, které opravdu chceme zpracovat, spolu s ID sloupcem.

Dále se již načtou data z této nového tabulky do *DBTableContent* objektu a převedou se pomocí *WekaUtils* modulu na *Instances* objekt. Následuje vlastní nastavení dat do filtru a jeho spuštění.

Tím jsme získali nová, normalizovaná data a již je jen potřebujeme převést zpět do databáze. To zajistíme postupním procházením *Instances* objektu. V cyklu procházíme jednotlivé normalizované sloupce a načítáme si k nim enumeraci všech jejich řádků, kterou procházíme v druhém, vnitřním, cyklu. Jednotlivé hodnoty jednak ID sloupce a vlastní hodnoty se získávají z instance a v případě id sloupce z položky attributes, do které vlastní instance má pouze odkaz.
Výsledný SQL dotaz je bohužel pro každou položku zvlášť a je tedy časově náročné zpracovávat velké množství dat. Tento čas se podařilo zredukovat použitím metody na hromadné poslání SQL dotazů *executeBatch*. Nicméně i tak je tato operace časově náročná.

Druhá implementace je závislá na agregačních funkcích obou databází. Naštěstí obě podporují jak průměr, tak získání směrodatné odchylky a bylo tedy možné použít vzorec a znovu jediný UPDATE dotaz pro celý sloupec. Pro výpočet z-score platí vzorec, kde se původní hodnota odečítá od průměru celého sloupce a výsledná hodnota se podělí směrodatnou odchylkou.

7.4 Dolovací moduly

Tato část pojednává o úpravách dolovacích modulů. Změny nebyly nijak zásadní. Polovina modulů byla schopna po menších úpravách fungovat s novými databázovými systémy. Druhá polovina nutně potřebuje celé algoritmy přepsat s využitím alternativ. Například z implementací Weka knihovny nebo některých dalších.

Ve všech modulech, které jsou nyní podporovány i MySQL a PostgreSQL implementací byly nastaveny uvedené databáze do operace *getAllowedDatabases* v hlavní třídě rozšiřující *MiningPiece.*

7.4.1 Modul shlukování

Tento modul obsahuje část algoritmů, které jsou již naimplementovány nezávisle na Oracle databázi. Některé naopak závislé jsou. Bylo tedy potřeba vytvořit rozlišení, jaké algoritmy se mohou a jaké nemohou v nových databázích použít.

K tomuto účelu byla ve třídě *ClusterAlgorithmInfo* vytvořena abstraktní operace *isOracleOnly(),* kterou musí každý z implementovaných algoritmů definovat. Pokud metoda vrací false, tak se tento algoritmus zobrazí v nabídce při připojení k jiným databázovým systému než k Oracle.

Z tohoto modulu byly také extrahovány třídy *WekaDataset* a *WekaUtils,* resp. část starající se o načítání dat, do nového balíčku weka.utils ve weka-wrapper modulu.

7.4.2 Modul predikce v časových řadách

Zde bylo na všech místech nahrazeno DME připojení klasickým SQL připojením. Kromě toho se zde změnilo také nastavování typů pro vytvoření nové tabulky. Původní již nebylo možné po změnách do *DBTableColumn* třídy použít, jelikož využívalo k převodu smazanou operaci.

7.4.3 Moduly Genetického algoritmu a neuronové sítě

Tyto moduly se dočkaly pouze změn v připojení. Nyní používá *ConnectionInfoSQL* a *Connection* z klasického SQL. V několika případech bylo potřeba změnit konstruktory u jednotlivých tříd.

7.4.4 Ostatní moduly

V ostatních modulech došlo k výměně *ConnectionInfoDME* na jeho nové rozhraní *IConnectionInfoDME,* což platí pro moduly, které nyní nejsou použitelné v nových SŘBD.

7.4.5 Návrhy na úpravu nekompatibilních modulů

Pro algoritmy detekce odlehlých hodnot, dolování asociačních pravidel, regrese i klasifikace pomocí různých metod existují v knihovně Weka určité použitelné implementace. Takže by mělo být možné je použit.

Nicméně po prozkoumání těchto algoritmů se zdá, že nejsou minimálně tak pružné a nastavitelné jako implementace zprostředkovaná přes Oracle databázi. Bude tedy nejspíše nutné tyto implementace upravit, nebo zúžit možnosti nastavení pro jiné databázové systémy.

Jednotlivé moduly již mají přístup k zjištění, jaká databáze se právě používá, skrz *MiningPiece* a potažmo *Kernel* třídu, takže implementace se podle tohoto mohou zachovat.

8 Testování

Testování probíhalo současně s vývojem aplikace. Prvně jsem narazil na několik drobných problémů již při testování na původní implementaci systému.

Například při využití komponenty transformace vůbec nebylo funkční ořezávání. Pro každý vstup se tedy vracely původní hodnoty. Tato chyba byla způsobena kontrolou řetězce typu ořezávání, kteréž mělo první písmeno velkým písmem, přičemž referenční řetěz malým. Po opravení této triviální chyby jsem našel druhý problém a to ten, že se při použití pouze typu *Trim* ořezávání špatně vybírá název tabulky, který se má vrátit jako výsledek. Respektive se pro tento typ ořezávání nastavovala špatná tabulka, kam se výsledek ukládal. Toto bylo způsobeno nenastavováním prvního jména tabulky na null, respektive nastavování jen při proběhnutí *Winsorize* typu ořezávání. Ten samý problém s výslednými tabulkami měla i normalizace.

Při testování šlo především o zachování stejných výstupů, kterých systém dosahuje na Oracle databázi, také na dalších dvou. S mírnými odchylkami se mi tento cíl podařilo zvládnout. Při testování operací předzpracování dat jsem se také snažil co nejvíce snížit čas, který je potřeba na jejich provádění. Ve většině případů se mi tohoto podařilo dosáhnout a některé jsou subjektivně rychlejší, než původní implementace.

V rámci testování jsem využíval různé datové soubory, které jsem získal především ze studijních materiálů k předmětu ZZN.

9 Přidání podpory pro další SŘBD

Tato kapitola popisuje změny nutné pro zprovoznění další SŘBD na systému FIT-Miner.

Prvně je potřeba enumeraci *DatabaseType* rozšířit o název nové SŘBD, nastavit název třídy ovladače a zda SŘBD podporuje JDM, či nikoliv. Dalším krokem je vytvoření nového modulu, který bude obsahovat dvě třídy, které musí implementovat rozhraní *DatabaseDataTypes* a *IDatabaseAPI*. Následně se obě tyto implementace přidají do továrních tříd *DatabaseApiFactory* a *DatabaseDataTypesFactory*. Největší část práce spočívá v implementaci *IDatabaseAPI* a jeho operací.

Pokud se jedná o SŘBD využívající JDM, pak se v rámci jádra musí vytvořit nová třída implementující rozhraní *IConnectionInfoDME*. Tato implementace se poté přidá do tovární metody v třídě *DMEConnectionFactory*. Pokud podporu JDM nemá, tak se tento krok přeskočí.

Posledním krokem je povolení nových databází přes operaci *getAllowedDatabases* u jednotlivých dolovacích modulů a částí v jádru systému – tedy v každé třídě odvozené z třídy *MiningPiece*.

Nepovinně je možné vytvořit SQL skripty pro uložené procedury, které zajišťují rozdělení dat.

Následně stačí ve spuštěné aplikaci přidat ovladač pro nový SŘBD v části „Services“ a vytvořit nový projekt s vybráním nové SŘBD, která by měla být již k dispozici.

10 Závěr

Diplomová práce navazuje na práci Ing. Šebka. Vzdáleněji také na práci Ing. Krásného a Ing. Maderu, kteří systém FIT-Miner vyvíjeli. Náplní této práce bylo analyzovat současný stav systému, jeho modulů a částí, především z pohledu databázové závislosti. Na základě této analýzy byla vytvořena koncepce změn, které umožnily plně zprovoznit Fit-Miner systém na dalších SŘBD. Tato koncepce byla v rozsahu uvedeném v této práci realizována a spolu s intenzivním testováním bylo zajištěno zprovoznění podpory pro dva nové databázové systémy.

V rámci tvoření diplomové práce, zabrala podstatnou část právě analýza. Nešlo ovšem pouze o analýzu vlastního systému a jeho zdrojových kódů ale také dolování z dat, různých dolovacích úloh v rámci modulů a také fungování DMSL jakožto podstatné části celého systému pro popis dolovaní. V neposlední řadě to byla také analýza transformování a celkově předzpracování dat.

Při analýze dat bylo zjištěno, že některé moduly by musely být prakticky celé přepsány, jelikož jsou silně závislé na JDM/ODM, což by znamenalo mnohem více potřebného času. Tudíž ne všechny moduly nyní fungují na nových databázích. Ovšem podařilo se úspěšně vyřešit zprovoznění celého jádra aplikace spolu s polovinou dolovacích modulů, které nebylo potřeba kompletně předělávat. Jako další rozšíření projektu by bylo vhodné zbylé moduly převést také na nezávislou implementaci z nějaké dolovací knihovny, či vytvořit implementaci vlastní. Další vytvořené moduly by již mohly také počítat s tím, že by měli pracovat i na dalších SŘBD.

Implementace probíhala postupně, prvním krokem byla úprava struktur jádra a připojení. Spolu s tím se přidaly nové třídy, podle dřívějšího návrhu. Některé implementační detaily a koncepce bylo zapotřebí rozhodnout v rámci konzultací, které byly vždy přínosné. Implementace třetí databáze již byla méně obtížná, jelikož následovala po všech úpravách jádra a nebylo tedy třeba přidávat o moc více věcí, než samotnou implementaci databázového rozhraní a typů.

Při úpravě systému se zasahovalo do velké části systému. Jádra, větší části dolovacích modulů, grafického rozhraní a především do komunikace s databází a interpretace databázových typů.

Jelikož jsem se s dolováním z dat před touto diplomovou prací setkal pouze okrajově, tak prozkoumání problematiky, a analyzování jednotlivých mechanik a procesů v rámci dolování z dat bylo velice zajímavé a také přínosné.

Literatura

[1] Šebek, M.: Rozšíření funkcionality systému pro dolování z dat na platformě NetBeans. Diplomová práce, FIT VUT v Brně, Brno, 2009.

[2] Mader, P.: Dolovací moduly systému pro dolování z dat v prostředí Oracle. Diplomová práce, FIT VUT v Brne, Brno, 2009.

[3] Krásný, M.: Systém pro dolování z dat v prostředí Oracle. Diplomová práce, FIT VUT v Brně, Brno, 2008.

[4] Gálet, M.: Grafická nadstavba pro systém získávání znalostí. Diplomová práce, FIT VUT v Brně, Brno, 2007.

[5] Wikipedia: NetBeans – Wikipedia, The Free Encyclopedia. 2013, [Online; navštíveno 3. 12. 2013]. URL: http://en.wikipedia.org/wiki/NetBeans

[6] NetBeans: Co je NetBeans? 2013, [Online; navštíveno 3. 12. 2012]. URL: http://www.netbeans.org/index_cs.html

[7] RICHARDSON, W. Clay. *Professional Java® JDK® 6 Edition*. 6th ed. Hoboken: John Wiley, 2007. ISBN 978-047-0126-608.

[8] Oracle Database Concepts 10g Release 2 (10.2) - B14220-02, [Online; navštíveno 7. 2. 2013]. URL: http://www.oracle.com/pls/db102/to_pdf?pathname=server.102%2Fb14220.pdf&remark=portal+%28Getting+Started%29

[9] Wikipedia: MySQL – Wikipedia, The Free Encyclopedia. 2013, [Online; navštíveno 6. 2. 2013]. URL: http://cs.wikipedia.org/wiki/MySQL

[10] PostgreSQL: Documentation: 9.2: PostgreSQL 9.2.2 Documentation, [Online; navštíveno 7. 12. 2012]. URL: http://www.postgresql.org/docs/9.2/interactive/index.html

[11] Weka: Framework Presentation [Online; navštíveno 6. 2. 2013]. URL: http://www.cs.colorado.edu/~kena/classes/6448/f08/framework_presentations/weka.pdf

[12] Zendulka, J.; Bartík, V.; Lukáš, R.; aj.: Získávání znalostí z databází – studijní opora. FIT VUT v Brně, Brno, 2009.

[13] Kotásek, P.: DMSL: The Data Mining Specification Language. Dizertacní práce, FIT VUT v Brne, Brno, 2003, [Online; navštíveno 3. 2. 2013]. URL http://www.fit.vutbr.cz/~kotasekp/dmsl/

[14] BÖCK, Heiko. *Platforma NetBeans: podrobný průvodce programátora*. Vyd. 1. Brno: Computer Press, 2010, 320 s. ISBN 978-80-251-3116-9.

[15] LACKO, Luboslav. *Databáze: datové sklady, OLAP a dolování dat s příklady v Microsoft SQL Serveru a Oracle*. 1. vyd. Brno: Computer Press, 2003, 486 s. ISBN 80-722-6969-0.

[16] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.

Seznam použitých zkratek

ODM – Oracle Data Mining
JDM – Java Data Mining
SŘBD – Systém řízení báze dat
SQL – Structured Query Language
DMSL - Data Mining Specification Language

Seznam příloh